ビジュアルラーニング
C言語入門

Visual Learning Introduction of C

さかお まい 著

本書の使い方

セクションの解説内容のまとめを表示しています。

セクション名は、解説内容を示しています。

キーワードを表示しています。

解説内容の見出しです。

プログラミングで利用する命令文の構文などを解説しています。

Section 12

繰り返し処理

覚えておきたいキーワード
- for文
- while文
- do〜while文

繰り返し処理のことを「ループ」といいます。ループを行う制御文を利用すると、ある条件を満たす間、同じ処理を繰り返すことができます。ループを行う制御文には、for文やwhile文、do〜while文などがあり、目的に応じて適切なものを選択する必要があります。

1. for文の利用

■ for文とは…

「for文」は、特定の条件を満たす間、処理を繰り返す制御文です。一般に、カウンタと呼ばれる変数を使って、処理の内容や対象を切り替えながら特定の回数だけ処理を繰り返す目的でよく使われます。for文の書式は次のとおりです。

構文 for文

```
for (初期化の式; 継続条件; 増減の式) 命令文;
```

構文 for文(ブロックの利用)

```
for (初期化の式; 継続条件; 増減の式) {
    …… 繰り返したい処理 ……
}
```

(1) 初期化の式

カウンタの初期値を指定します。変数の宣言も同時に行うことができます。

(2) 継続条件

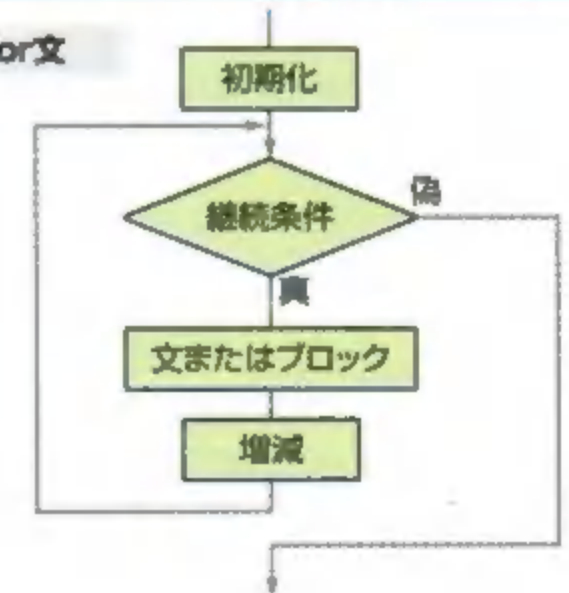
処理を繰り返すための条件を指定します。

(3) 増減の式

カウンタの増減の方法を指定します。

図1 for文

3 制御構文

72

本書では、基本的に2〜8ページ単位で1つのセクションを構成し、セクションの先頭には、そのセクションで解説する内容の概要とキーワードをまとめています。

各セクションでは、「構文」「サンプルプログラム」「実行結果画面」を順に目で追うことでC言語のプログラミングを学習できるようにしています。

操作が必要な場面では、操作の流れに番号を付けて示すことで、手順を追いやすくしています。

また、そのセクションの補足説明や参照事項を「コラム」として解説しています。

Visual Learning Introduction of C
ビジュアルラーニング
C言語
入門
さかお まい 著

●本書に掲載したサンプルプログラムについて
本書の中で使用したサンプルプログラムは、以下のURLからダウンロードすることができます。
http://www.x-media.co.jp/xbook/sample/

●本書の動作環境について
本書は、以下の環境で正常に動作することを確認しております。
・Windows XP Home Edition SP2 / Professional Edition SP2
・Borland C++ Compiler 5.5

はじめに

「ビジュアルラーニング」シリーズは、「ひとにやさしく情報を伝える」学習書です！
エクスメディアは、つねに読みやすくもっとわかりやすい学習書に挑戦します！

「ビジュアルラーニング」シリーズは、「ひとにやさしく情報を伝える」情報デザイン企業エクスメディアが制作する、「フルカラー」のメリットを最大限に生かして、豊富な図解を盛り込んだ「目に見えてよくわかる学習書」です。

[ビジュアルラーニングシリーズの特長]

- 視覚的な解説を有効活用し、専門知識が自然と理解できるように工夫しています。
- 読者が抱く小さな疑問も予測して、できるだけていねいに解説しています。
- 「広開本」製本を採用し、「開いたら閉じにくい書籍」を実現しています。

『ビジュアルラーニングC言語入門』の執筆にあたって

C言語は、世界でもっとも普及しているプログラミング言語です。いまやC言語の影響を受けていないプログラミング言語はほとんどなく、C言語はもはやプログラミング言語の「世界標準」といえるでしょう。

C言語に代わる新しい言語（C++やJavaなど）がいくつも登場して久しいですが、むしろC言語を理解することの重要度は増しているといえます。C++やJavaの概念を学習するためにはC言語の基礎的な文法が前提条件として必須であったり、「C言語を理解している人向け」のプログラミング書籍が世の中にはあふれていたりと、まずC言語を理解していないと次のステップへはなかなか進めないのが現状です。

本書では、C言語をプログラミング学習の第一歩として本書を選択した方の手助けをするべく、ゼロからの解説を目指して、次のような点に配慮して執筆しました。

- ソースファイルや概念を図解することで、ストレスなく読み進めることができる！
- 章のまとめや練習問題で復習することで、学習内容をより深く理解することができる！
- 操作手順や実行結果が、画面で表示される内容と同じなので安心できる！

最後に、本書がC言語を使いこなしたいみなさんの一助となれば幸いです。

2005年7月

さかお まい

Contents ＞＞＞ ビジュアルラーニング C言語入門

第2章 変数と演算子

第3章 制御構文

第4章 配列

第5章 関数

第6章 ポインタ

第7章 構造体

第8章 標準入出力ライブラリ

第9章 ファイル入出力

第10章 プリプロセッサ命令

第1章

Visual Learning Introduction of C

初めての
Cプログラミング

Section 1 C言語とは
Section 2 プログラム作成の流れ
Section 3 ソースファイルの作成
Section 4 コンパイルと実行
Section 5 コーディングの注意点とよくあるエラー

C言語とは

C言語は、世界中でもっとも利用されているプログラミング言語のひとつです。またC言語は、C++やJavaなどの新しいプログラミング言語のもととなったものなので、C言語を学習することは、これら新しいプログラミング言語の基礎を学習することにもなります。

1. C言語の概要

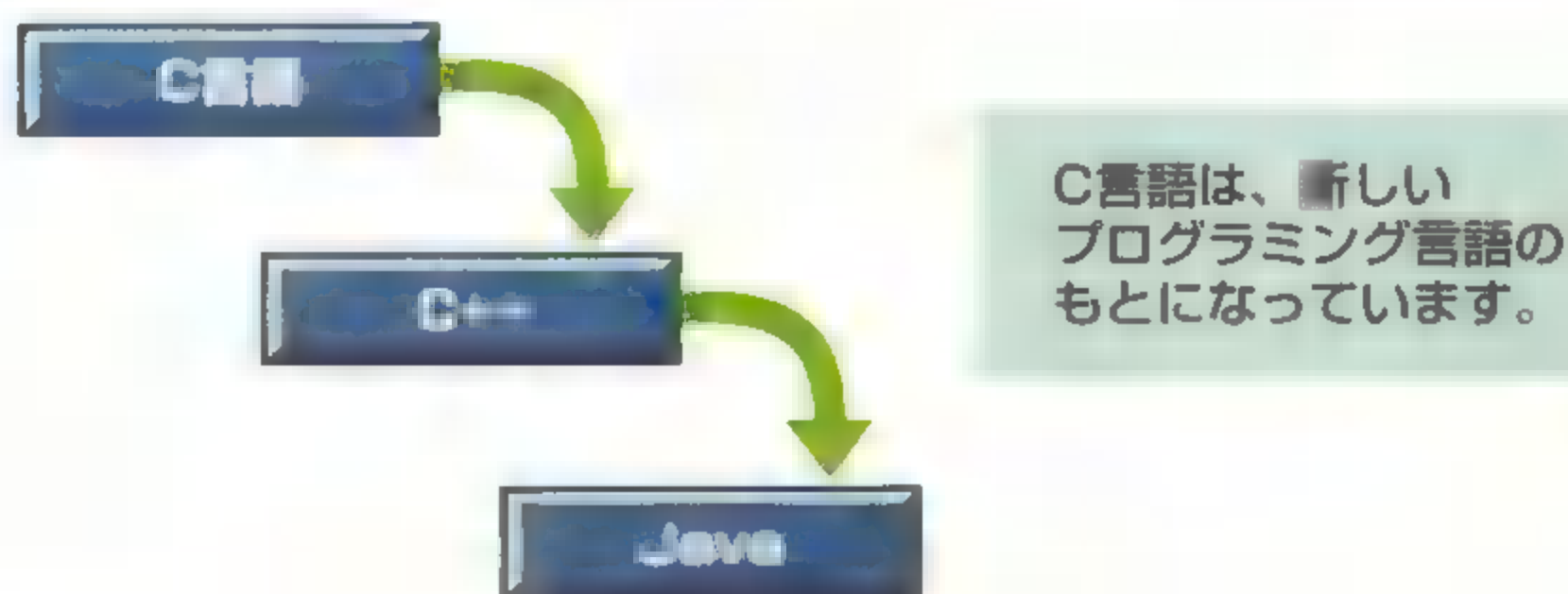
■ C言語とは…

C言語は、1972年頃に設計された「**プログラミング言語**」のひとつです。C言語は文法がわかりやすく、さまざまな用途に活用できたため幅広く普及しました。

現在主流になっているプログラミング言語に「**C++**」や「**Java**」などがありますが、これらの基礎的な部分はC言語の文法を踏襲しています。「C++」は、C言語に「オブジェクト指向」という考え方を加えて設計されたプログラミング言語であり、「Java」はC++をベースに、既存の言語の欠点を補う形で設計し直された言語です。

つまりC言語は、C++やJavaなどのもとになった言語だといえます。

図1　C言語、C++、Java

C言語は、新しいプログラミング言語のもとになっています。

■ C言語の特徴

C言語は、次のような特徴を持っています。

(1) 移植性が高い

ハードウェアやOSに固有の言語（アセンブラなど）とは違い、C言語には「**ANSI C**」といわれる**標準規格**が存在します。そのため、その規格にもとづいてプログラムを記述すれば、ハー

ドウェアやOSが異なっても、そのプログラムのソースファイルをそのまま別のハードウェアやOSで利用することができます。

ANSI C規格を日本語に翻訳したものを「**JIS C**」といいます。ANSI CとJIS Cの内容は同じものです。

(2) 文法が簡単で柔軟な処理が可能

C言語の文法は覚えることが少なく、記述ルールは簡単です。それにもかかわらずアセンブラのようにハードウェアに近い部分まで柔軟に処理できます。

(3) サイズが小さく、高速なプログラムを作成可能

C言語で作成したプログラムは、コンピュータが直接実行できる形式に変換するため、比較的サイズが小さく、処理が高速です。

(4) 用途が広い

C言語はアプリケーションに限らず、コンパイラなどのシステム寄りのプログラムや、機械やネットワークなどの制御系のプログラム、ゲームやグラフィックス関係のプログラムといったさまざまな用途に使われています。

2. プログラムが動作するしくみ

C言語では、目的の「プラットフォーム」(OSやハードウェアを組み合わせたコンピュータ環境)に合わせたプログラムを作成できます。

C言語で記述したプログラムファイルのことを「ソースコード」あるいは「ソースファイル」などといいます。ただし、ソースコードのままではコンピュータはプログラムを実行することができません。ソースコードを目的のプラットフォームで実行できる形に変換する必要があります。この手続きを「コンパイル」、コンパイルを行うソフトウェアのことを「コンパイラ」といいます。

本書では、Windows上で実行できるプログラムを作成する方法を解説します。

図2 プログラムの作成

Section 2

プログラム作成の流れ

覚えておきたいキーワード

- コーディング
- バグ
- コンパイラ

C言語のプログラミングでは、まずソースコードをコンパイルし、コンピュータで実行できる形に変換します。コンパイラにはいくつかの種類があり、コンパイルを「コマンドプロンプト」で実行するものと「GUI」で実行するものがあります。

1. C言語のプログラミング

■ プログラミングの流れ

コンピュータで実行するプログラムを作成することを「**プログラミング**」といいます。C言語では、次のような流れでプログラミングを行います。

(1) コーディング

コンピュータに行わせたい処理を、人間に理解できるC言語で記述します。これを「**コーディング**」といいます。

(2) コンパイル

作成したソースコードを、コンピュータが理解できる形式に変換します。これを「**コンパイル**」といい、コンパイルによって「**オブジェクトファイル**」が作成されます。

(3) リンク

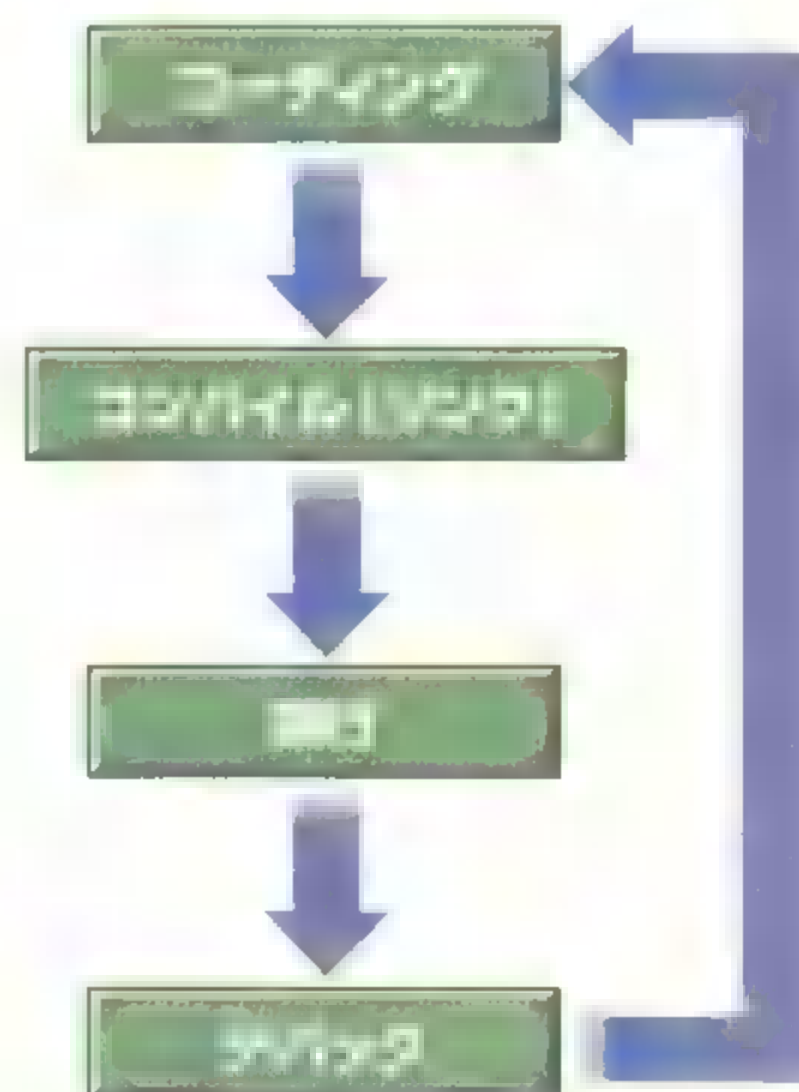
オブジェクトファイルに、プログラムの実行に必要なファイルを結びつけて実行可能形式のプログラムを作成します。これを「**リンク**」といい、リンクを実行するプログラムを「**リンカ**」といいます。ただし、たいていのコンパイラでは、コンパイルをする際にあわせてリンクまで行うように設定されています。

図1 プログラミングの流れ

(4) 実行

作成した実行可能形式のプログラムを実行します。

(5) デバッグ

プログラムの動作に、意図しない動作や誤りがない

かを確認します。誤りが発見された場合は、コーディングに戻り間違いを修正します。

このプログラムの間違いを「**バグ**」、ソースコードのどこに間違いがあるか探す手順を「**デバッグ**」といいます。

■ C言語の開発環境について

C言語でプログラミングを行うには、**コンパイラ**をコンピュータにインストールする必要があります。Windows用のコンパイラのほとんどは、「**コマンドプロンプト**」でコマンドを打ち込むことによって動作します。このように、文字ベースのプログラムの実行環境を「**CUI**（Character User Interface）」と呼びます。

代表的なCUI環境のコンパイラのうち無償で入手できるものには、次のようなものがあります。

表1　代表的なCUIコンパイラ

コンパイラ	URL
Borland C++ Compiler 5.5	http://www.borland.co.jp/cppbuilder/freecompiler/
Microsoft .NET Framework SDK	http://www.microsoft.com/japan/msdn/netframework/downloads/
Digital Mars C/C++ Compiler	http://www.digitalmars.com/（英語）
GCC	http://gcc.gnu.org/（英語）

本書では「Borland C++ Compiler 5.5」を利用して解説を行います。インストール方法の詳細についてはP.228の付録1を参照してください。

また、Windows上で動作するプログラムの中でコーディングからコンパイル、実行までを行える「**統合開発環境**（IDE、Integrated Development Environment）」という製品を購入して利用することもできます。ウィンドウやアイコンなどを使った、グラフィカルなプログラムの実行環境を「**GUI**（Graphical User Interface）」と呼びます。

代表的なIDEには、次のようなものがあります。

表2　代表的なIDE

IDE	URL
Microsoft Visual Studio	http://www.microsoft.com/japan/msdn/vstudio/
Borland C++Builder	http://www.borland.co.jp/cppbuilder/

ソースファイルの作成

覚えておきたいキーワード

- テキストエディタ
- インデント
- コメント

C言語の特徴を理解したら、実際にプログラムを作成してみましょう。まず、ソースファイルを作成します。ソースファイルは、テキストエディタと呼ばれるアプリケーションを利用して作成します。市販のテキストエディタを利用すると、効率的なコーディングが可能になります。

1. テキストエディタ

「**テキストエディタ**」とは、その名のとおり「文字編集」を行うアプリケーションのことです。たとえば、Windowsにあらかじめ付属している「メモ帳」などのことをいいます。

C言語のソースファイルを作成するには、テキストエディタが必要です。Windowsにあらかじめ付属しているメモ帳でもソースファイルを作成することができますが、市販のテキストエディタを利用するのが効率的で、一般的です。これらのテキストエディタは、主にインターネットのWebサイトからダウンロードして入手することができます。

代表的なテキストエディタに、シェアウェア（有料のソフト）の「**秀丸エディタ**」や、フリーウェア（無料のソフト）の「**サクラエディタ**」などがあります。これらのエディタは、それぞれ下記のWebページからダウンロードすることができます。

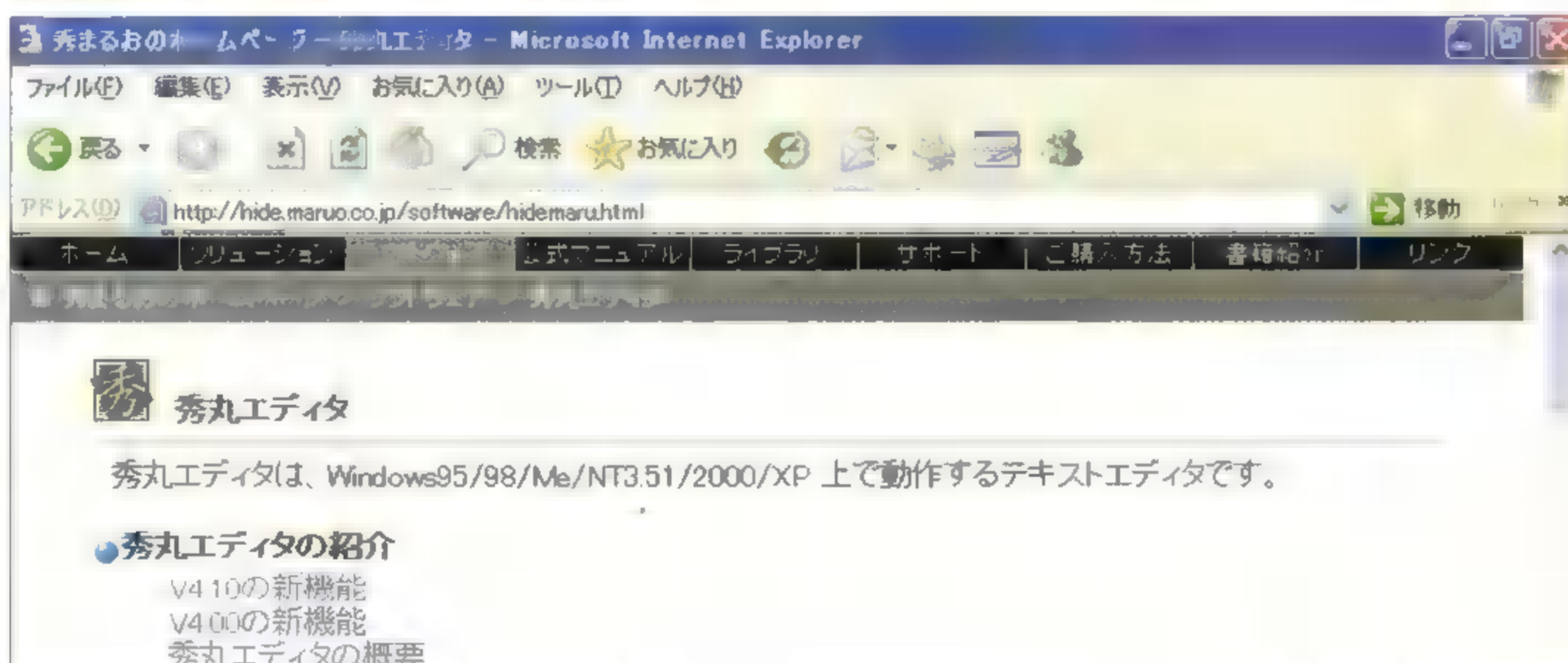
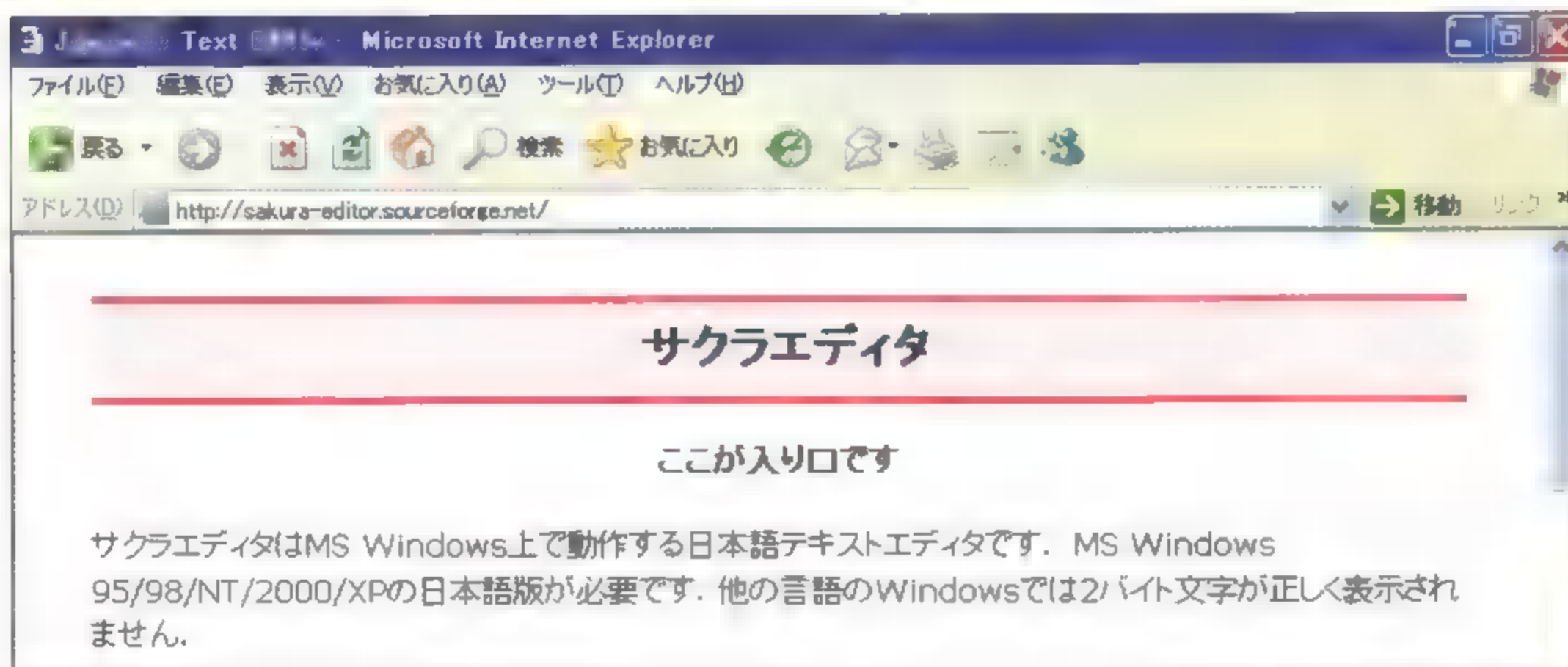
図1　秀丸エディタのWebページ

Ⓒ 秀まるお氏

図2　サクラエディタのWebページ

http://sakura-editor.sourceforge.net/
© テキストエディタオープンソースプロジェクト

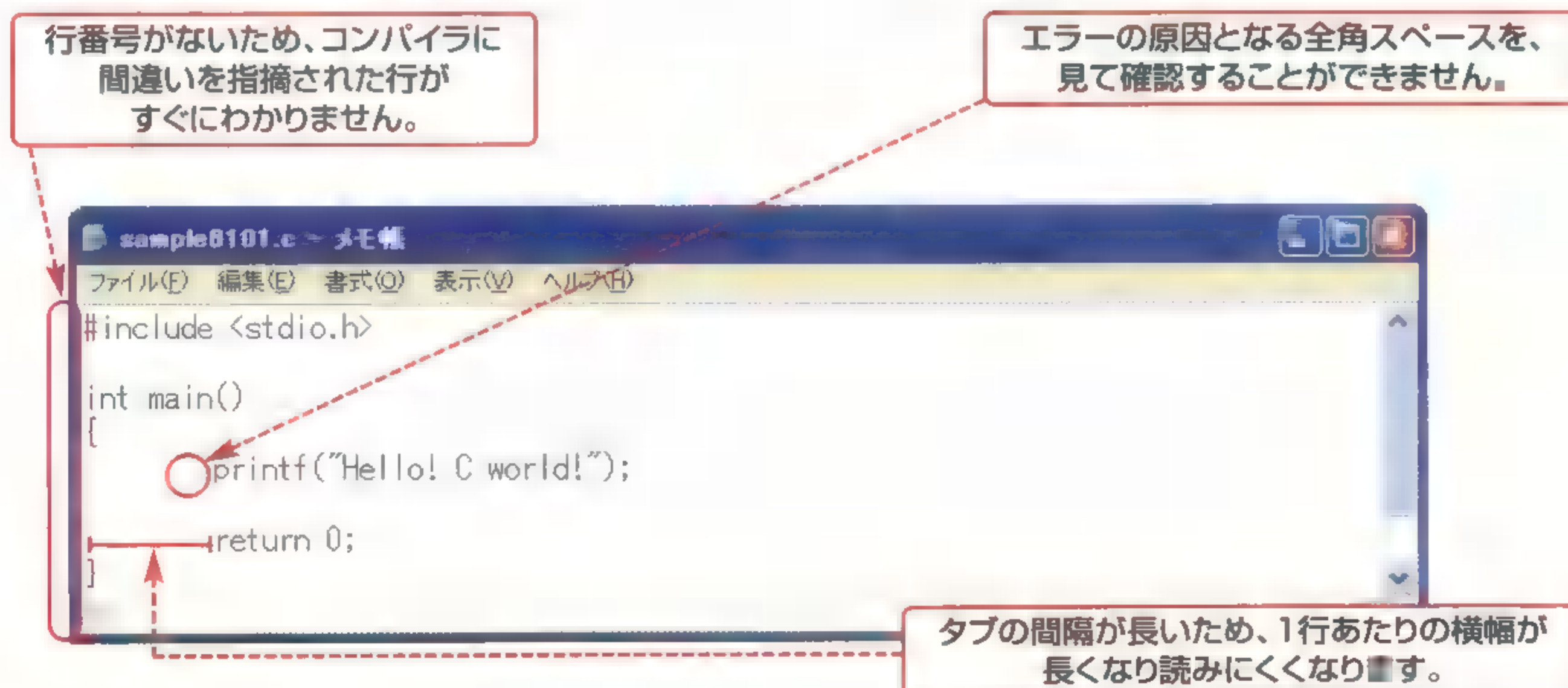
2. テキストエディタの主な機能

■ メモ帳の不便な点

Windowsに付属しているメモ帳は、文字を編集する上で必要最低限の機能しか搭載されていません。そのため、プログラミングを行う上で次のような不便な点があげられます。

(1) コンパイルの際に行番号で指摘されるエラーの場所がわかりにくい。

(2) ソースコードで使用することができない「全角スペース」を見分けられない。

(3) タブの文字送りの数が8文字に固定されているため、画面が横に広くなって読みにくい。

3. ソースコードの入力

■ C言語の基本的なルール

テキストエディタの準備が整ったら、いよいよソースコードを入力してみましょう。C言語のもっとも基本的な規則には、次のようなものがあります。

(1) 英数字や記号は、すべて半角文字で入力する

文字列として表示したいなどの特別な場合を除き、全角文字は使用しません。

(2) 英字の大文字と小文字は、別の文字として扱う

たとえば「main」と「Main」とは、まったく別の名前として扱われます。

(3) 単語やカッコの間の空白部分は、半角スペースかタブを入力する

半角スペースやタブを何文字入れても問題ありませんが、全角スペースは利用できません。

(4) 命令文の文末に「;(セミコロン)」を付ける

命令文は、改行があってもそこで終わりになりません。命令文の終わりを示すためにはセミコロンを入力します。

これらの点に気をつけて、とりあえず以下に示すとおりにソースコードを入力してみてください。意味はまったくわからなくても問題はありません。後続の章で徐々に解説していきます。

[illegible] 基本的なプログラム

```
01  #include <stdio.h>
02  int main()
03  {
04      printf("Hello! C world!");
05      return 0;
06  }
```

4. サンプルプログラム「hello.c」の概要

サンプルプログラム「hello.c」の概要について、簡単に触れておきます。詳細はそれぞれ後で解説するので、そちらを参照してください。

(1) main()関数

C言語のソースコードには、必ず**main()**関数が必要です。C言語で作られたプログラムは、

必ずこの**main()**関数から処理が始まります。

処理は**main()**関数内の上から順番に行われ、最後の「}」でプログラムが終了します。

なお、「関数」の意味は第5章で詳しく説明します。

```
int main()
{

}
```

この部分が上から順番に実行されます。

(2) printf()関数

printf()関数は、画面に文字列を表示するための関数で、C言語に標準で用意されています。ここでは「"（ダブルクォーテーション）」で囲んだ文字列が画面に表示されます。

printf()関数のさらに詳しい使い方は、第2章および第8章で解説します。

(3) #include文

#include文は、「このソースファイルに、他のソースファイルの機能を取り込みたい」場合に記述します。この列では「**printf()**関数」を利用したいので「stdio.h」というファイルを読み込みます。このファイルの読み込み作業のことを「**インクルード**」と呼びます。

#include文について、詳しくは第10章で解説します。ここでは「**printf()**関数が使いたいので書いてある文」ということだけ認識してください。

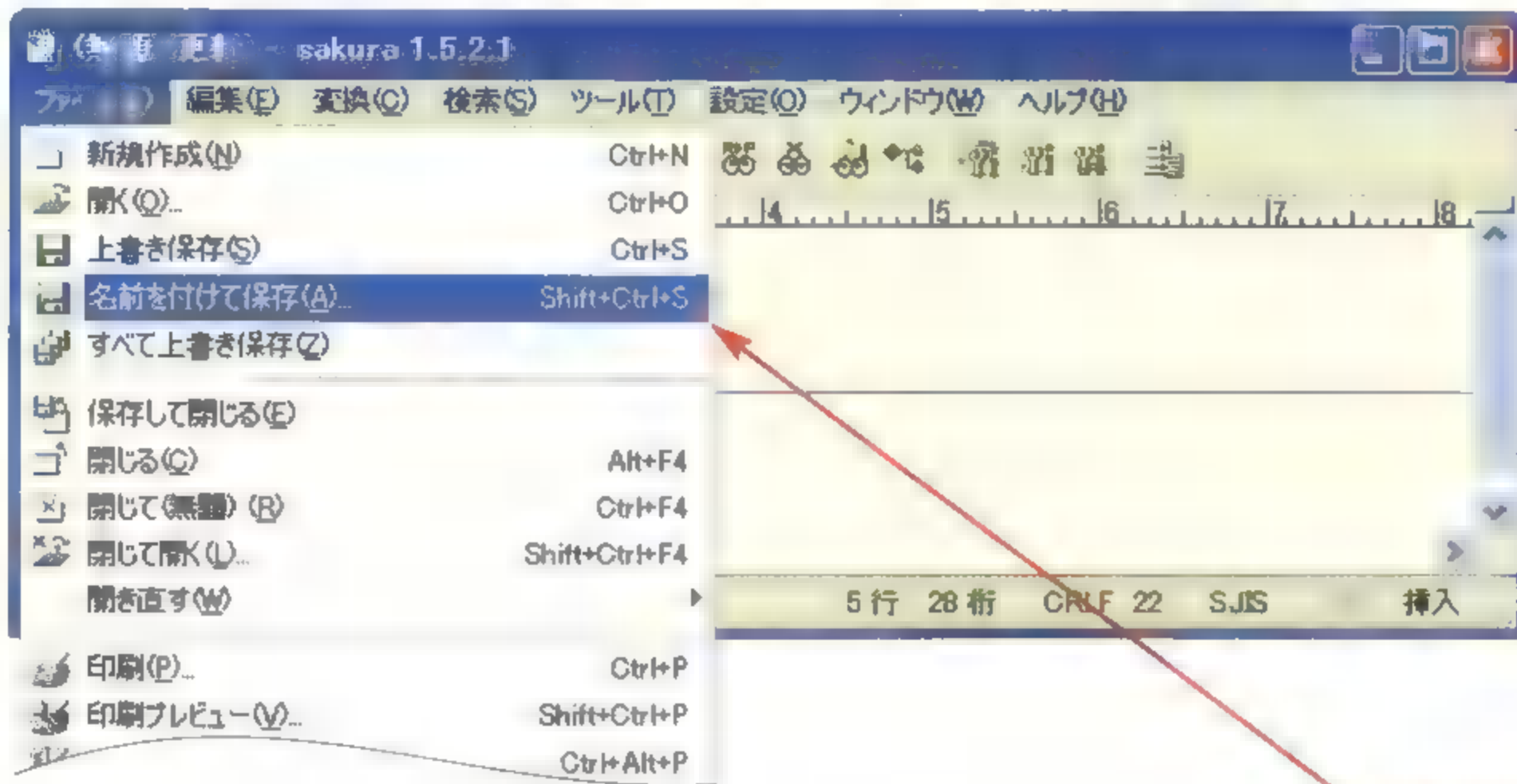
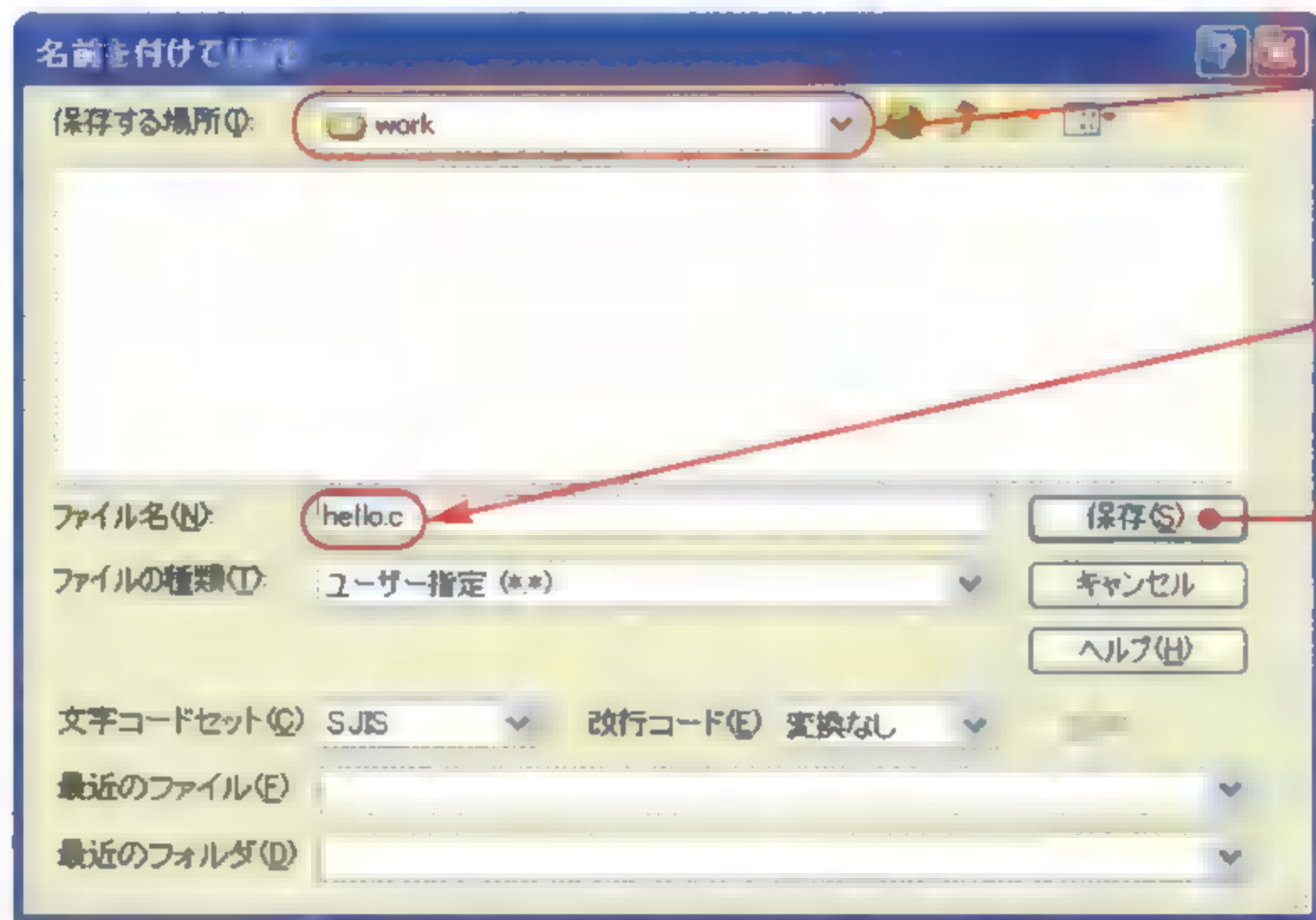
(4)「{」から「}」まで

「{」と「}」で囲まれている部分を「**ブロック**」といいます。このサンプルを例にとると、「**main()**関数の始まりと終わりを示している」という意味です。「{」によってブロックを始めたら、必ず「}」によって閉じなければいけません。

5. ソースファイルの保存

ソースコードを入力したら、保存してソースファイルを作成します。保存場所には、あらかじめ作成しておいた作業ディレクトリを指定します。本書では、「C:¥work」フォルダを作業ディレクトリとして使用し、このフォルダににソースファイルを保存します。

ソースファイル名には英数字で好きな名前を付けられます。ただし、**拡張子**を「.c」にしなければなりません。

ここでは、先ほど入力したソースコードを「hello.c」というファイル名で保存します。

6. 読みやすいソースコード

■ ソースコードを読みやすく記述する理由

ソースコードは、できるだけ読みやすく記述するように心がける必要があります。読みにくいソースコードは、他人だけでなく、書いた本人ですら後で読み返して意味がわからなくなるからです。何をしたいプログラムなのかパッと見ただけではわからないのでは、効率が悪いだけでなく、バグが発生しやすくなります。

Sample0102.c　読みにくいソースコード

```
01  int main(){
02  printf(
03  "Hello! C world!");
04  return 0;
05  }
```

■ インデント

タブ文字を利用して**インデント**を行うと、ソースコードが見やすくなります。また、テキストエディタによっては「オートインデント」という機能があり、自動的にインデントを行う場合があります。

インデントや{　}の付け方は一般的な「流儀」がいくつかあります。たとえば{　}の付け方では、次のように、文に続けて記述する流儀もあります。

```
int main() {
```

インデントや{　}の付け方については、この先いろいろなソースコードを読んで、読みやすく自分に合っていると思うものを選択するとよいでしょう。

■ コメント

ソースコードには、「**コメント**」と呼ばれる「プログラマのメモ」を記述することができます。コメントは、処理を行うコードとして認識されません。何を書いても、コンパイラによって無視されます。ソースコードの処理の目的や内容などをコメントとして記述しておくと、後で自分や他人がコードを解析するときに便利です。コメントは、次のように記述します。

構文　コメント文

```
/* コメント */
```

「/*」と「*/」で囲まれたすべての部分がコメントになります。複数行になってもかまいません。コメントを利用したプログラムは、次のようになります。

```
/* 画面に文字を表示する */
printf("Hello! C world!");
```

Section 4

コンパイルと実行

覚えておきたいキーワード
- ディレクトリ
- パス
- プログラムの実行

ソースファイルを作成できたら、それをコンパイルして実行してみましょう。コンパイルと実行は、コマンドプロンプト画面でコマンドを入力して行います。コンピュータへの指示は、すべてコマンドプロンプト上でキーボードから入力するコマンドで行います。

1. コマンドプロンプトの利用

コマンドプロンプトとは...

ソースファイルのコンパイルとプログラムの実行は、「**コマンドプロンプト**」と呼ばれる機能を使って行います。

コマンドプロンプトとは、Windowsの先代のOSである「**MS-DOS**（エムエスドス）」で行っていた操作をWindows上で行うための機能です。Windowsのように操作対象が絵で表現されるGUI（Graphical User Interface）ではなく、操作対象が文字で表現され、コマンドと呼ばれる命令をキーボードで入力して操作するCUI（Character User Interface）で操作することが大きな特徴です。なお、「MS-DOSプロンプト」や「DOSプロンプト」とも呼ばれます。

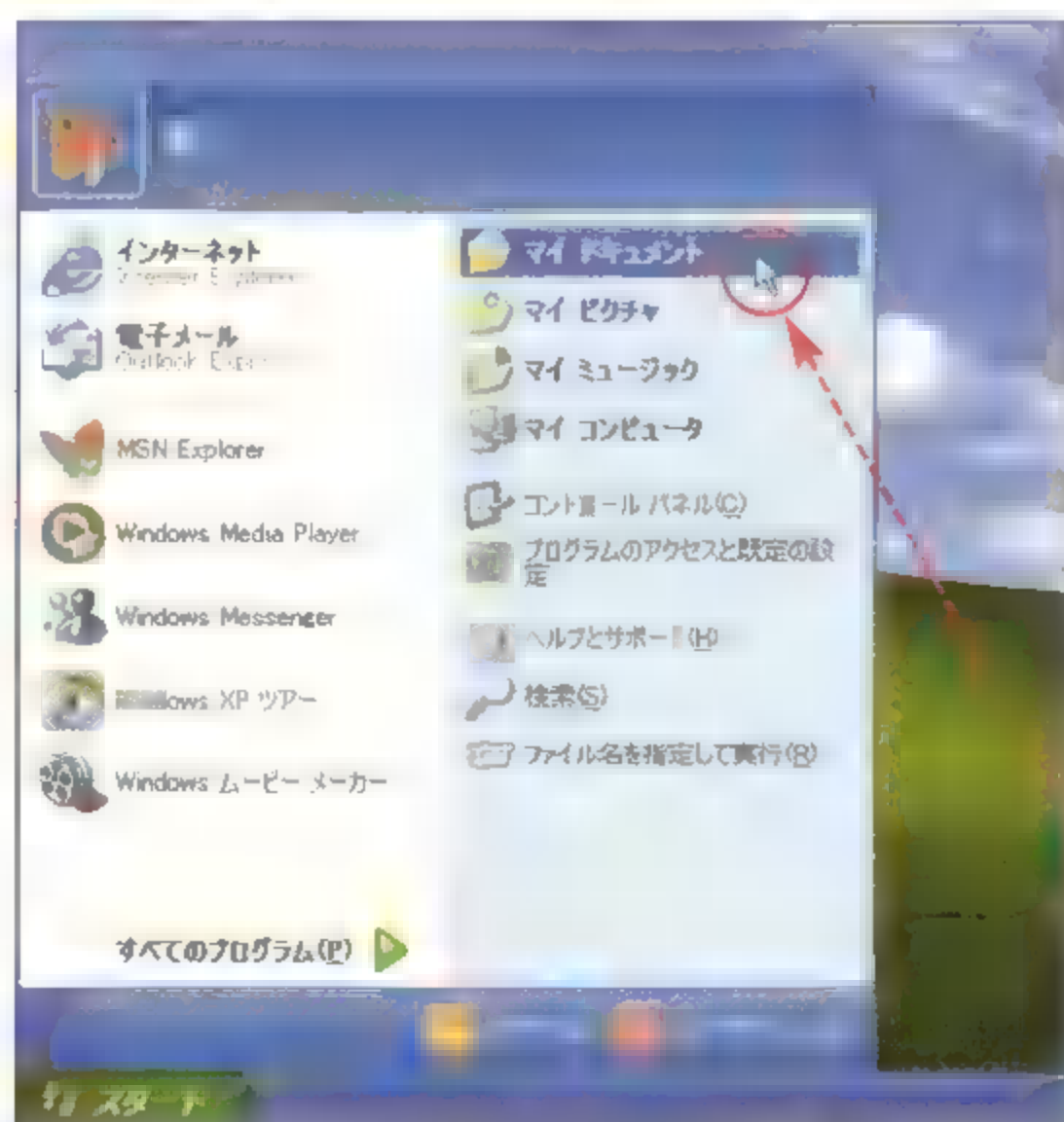
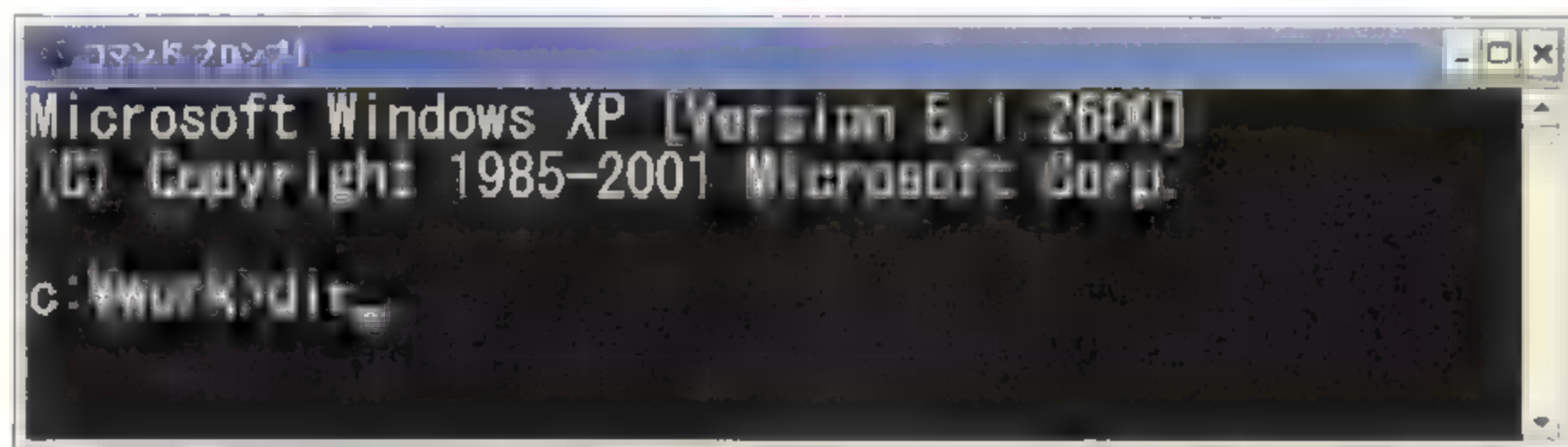
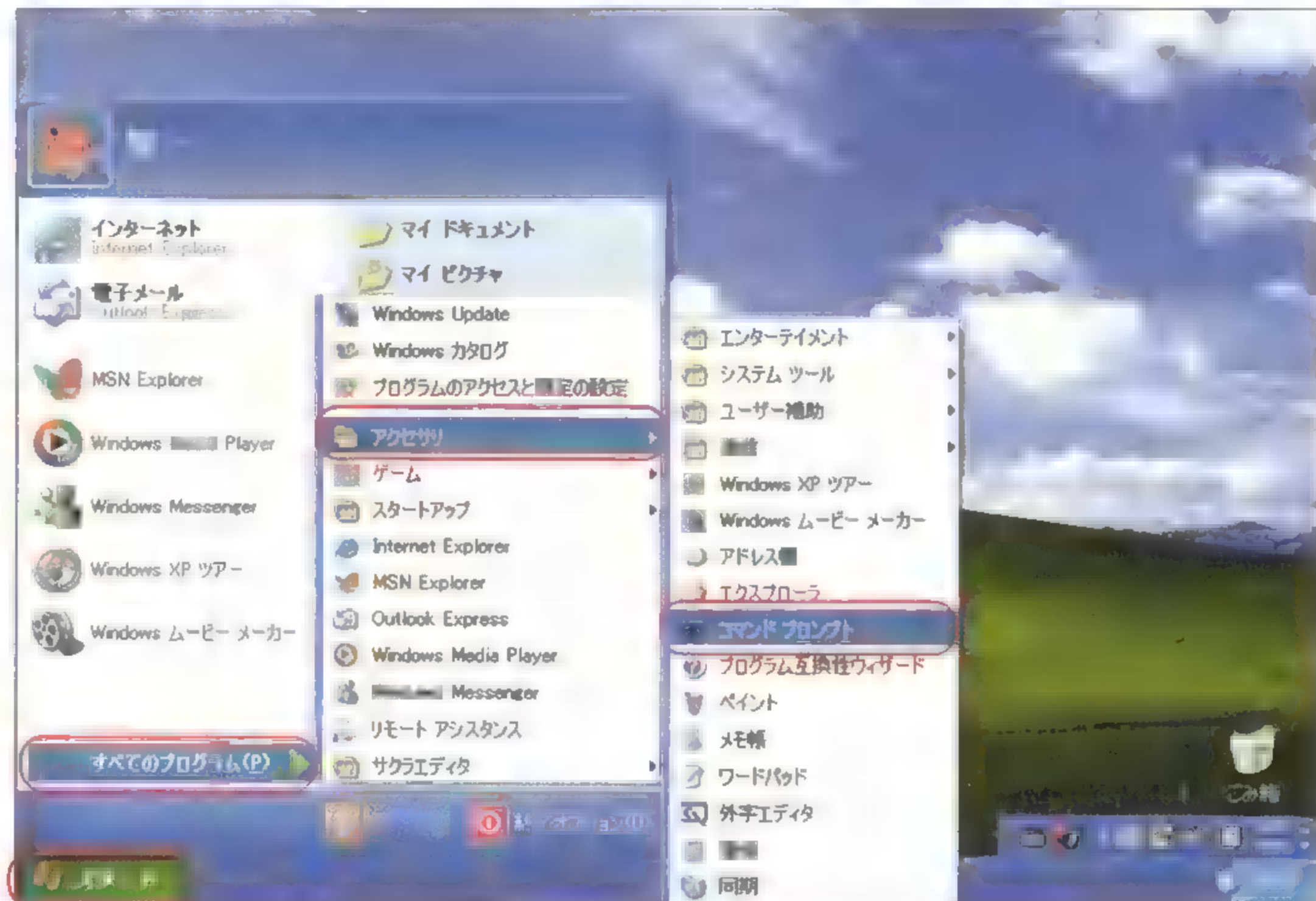
図1 Windows XP(GUI)

主にマウスを利用して操作します。

図2 コマンドプロンプト(CUI)

キーボードからコマンドを入力して操作します。

■ コマンドプロンプトの起動

コマンドプロンプトを起動するには、[スタート] メニューで [すべてのプログラム] をポイントし、[アクセサリ] をポイントして、[コマンドプロンプト] を選択します。

図3 コマンドプロンプトの起動

また、[スタート] メニューで [ファイル名を指定して実行] を選択すると表示されるダイアログボックスで、<名前>に「**cmd**」と入力して<OK>ボタンをクリックしても、コマンドプロンプトを起動することができます。

■ コマンドプロンプトの終了

コマンドプロンプトを終了するには、ウィンドウ上部の<閉じる>ボタン ✕ をクリックするか、コマンドプロンプトで「**exit**」と入力して Enter を押します。

2. ディレクトリとパス

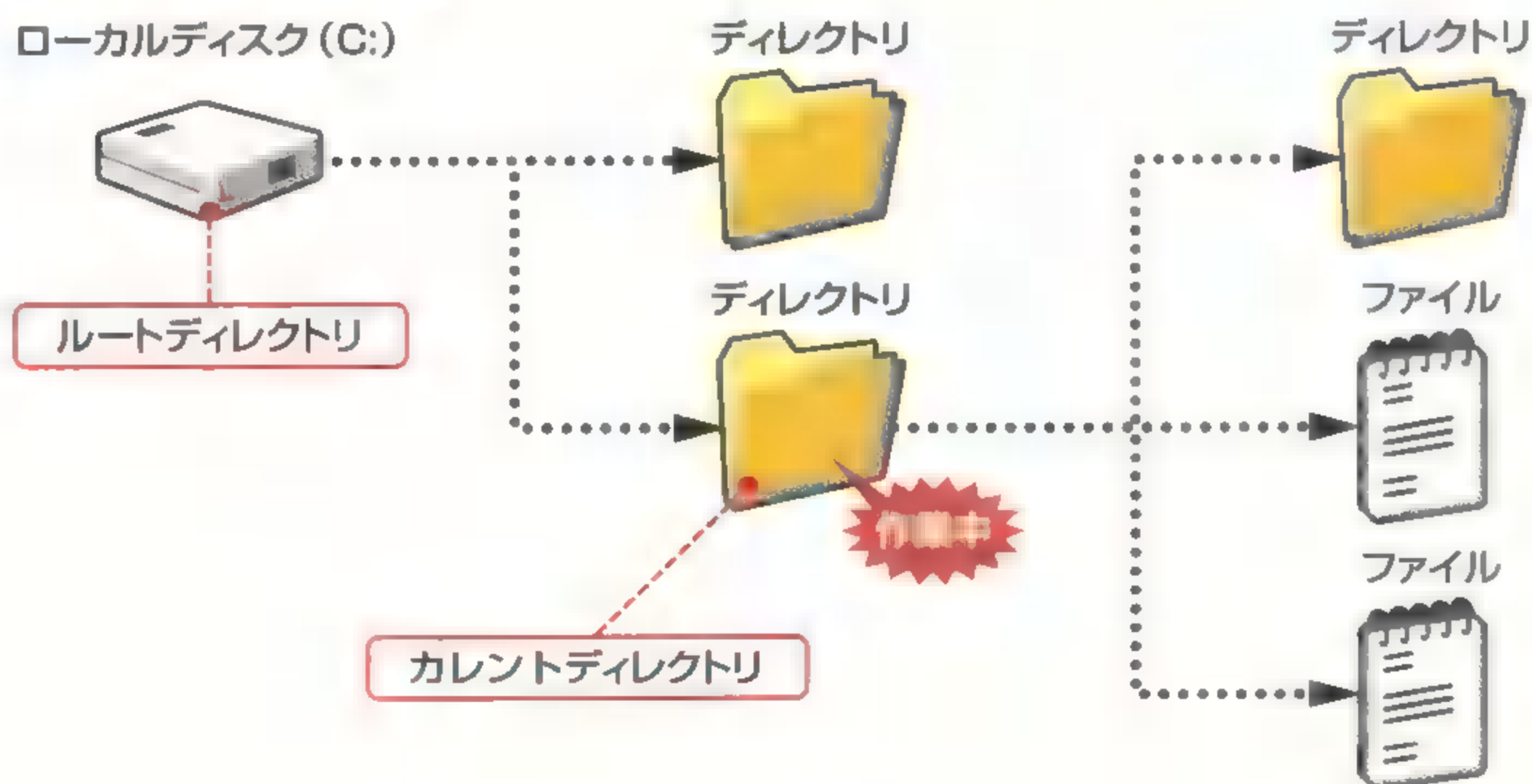
■ 場所やファイルの指定

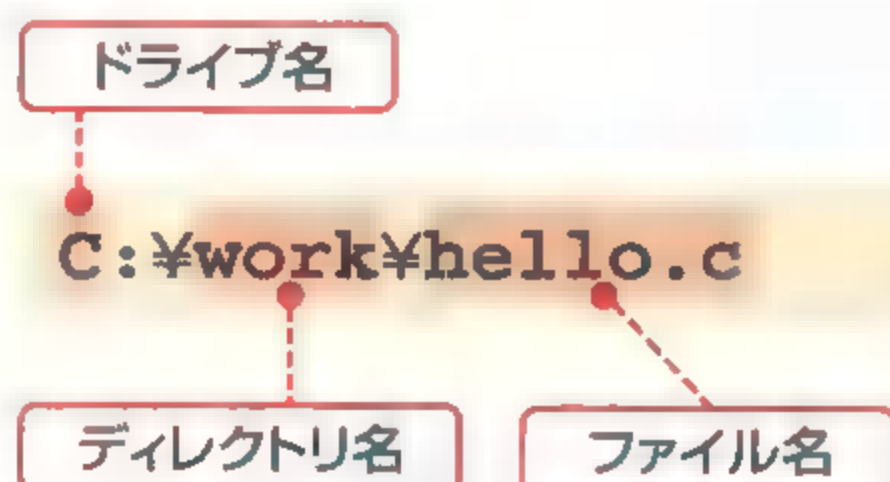
Windowsでは、マウスを利用して現在作業しているフォルダを移動したり、ファイルをコピーしたりします。しかし、コマンドプロンプトでは、すべての操作をキーボードから行うため、現在作業している場所や目的のファイルも、文字列で表現する必要があります。これらを表現するのに、ファイル名やディレクトリ、パスが利用されます。

■ ディレクトリとは…

「**ディレクトリ**」とは、ファイルをまとめて整理する場所のことで、フォルダと同じ意味です。一般にMac OSやWindowsでは「フォルダ」、UNIXやMS-DOSでは「ディレクトリ」と呼びます。従って、コマンドプロンプトを利用する場合にはディレクトリと呼びます。また、ハードディスクの最上位のディレクトリを「**ルートディレクトリ**」、現在作業しているディレクトリを「**カレントディレクトリ**」と呼びます。

図4 ディレクトリとファイル

ローカルディスク(C:)
ルートディレクトリ
ディレクトリ
ディレクトリ
作業中
カレントディレクトリ
ディレクトリ
ファイル
ファイル

■ パスとは…

「**パス**」とは、ファイルやディレクトリの場所を表す文字列のことです。ドライブ名とディレクトリ名、ファイル名から構成され、次のように記述します。なお、「**ドライブ名**」とは、OSがハードディスクやフロッピーディスクなどに割り当てる名前のことで、Windowsでは1文字の英字で表現されます。たとえば、<ローカルディスク(C:)>はCドライブにあたります。

パスは上の階層から順番に記述していき、ドライブ名の直後には「:(コロン)」を入力します。また、ドライブ名とディレクトリ名、ファイル名はすべて「¥」で区切ります。「ドライブ名」「:(コロン)」「¥」は、すべて半角文字で入力します。

なお、このようにルートディレクトリから目的のディレクトリ(ファイル)まで順番に記述したパスを「**絶対パス**」と呼びます。

■ 相対パス

絶対パスに対して、カレントディレクトリを基準に目的のディレクトリ(ファイル)の場所を記述したパスを「**相対パス**」と呼びます。相対パスでは、カレントディレクトリを「.(ピリオド1つ)」、1つ上の階層のディレクトリを「..(ピリオド2つ)」で表します。また、カレントディレクトリを表す「.」は省略することができます。

たとえば、「hello.c」ファイルを指定する場合、ファイルがカレントディレクトリにある場合は、

```
.¥hello.c   または   hello.c
```

と記述します。また、カレントディレクトリの1つ上の階層にある場合は、

```
..¥hello.c
```

と記述します。

3. コマンドプロンプトの構成要素

コマンドプロンプトは、次のような要素で構成されます。

(1) プロンプト

入力できる状態であることをユーザーに示す特殊な文字のことで、「カレントディレクトリの絶対パス表示」と「>」が組み合わされて表示されます。

(2) コマンド

さまざまな操作を行うための命令です。

(3) 引数

コマンドで行う操作の対象などを記述します。コマンドの種類によって必要とする引数の数は異なり、引数がない場合もあります。コマンドおよび各引数は、1つ以上の半角スペースで区切ります。

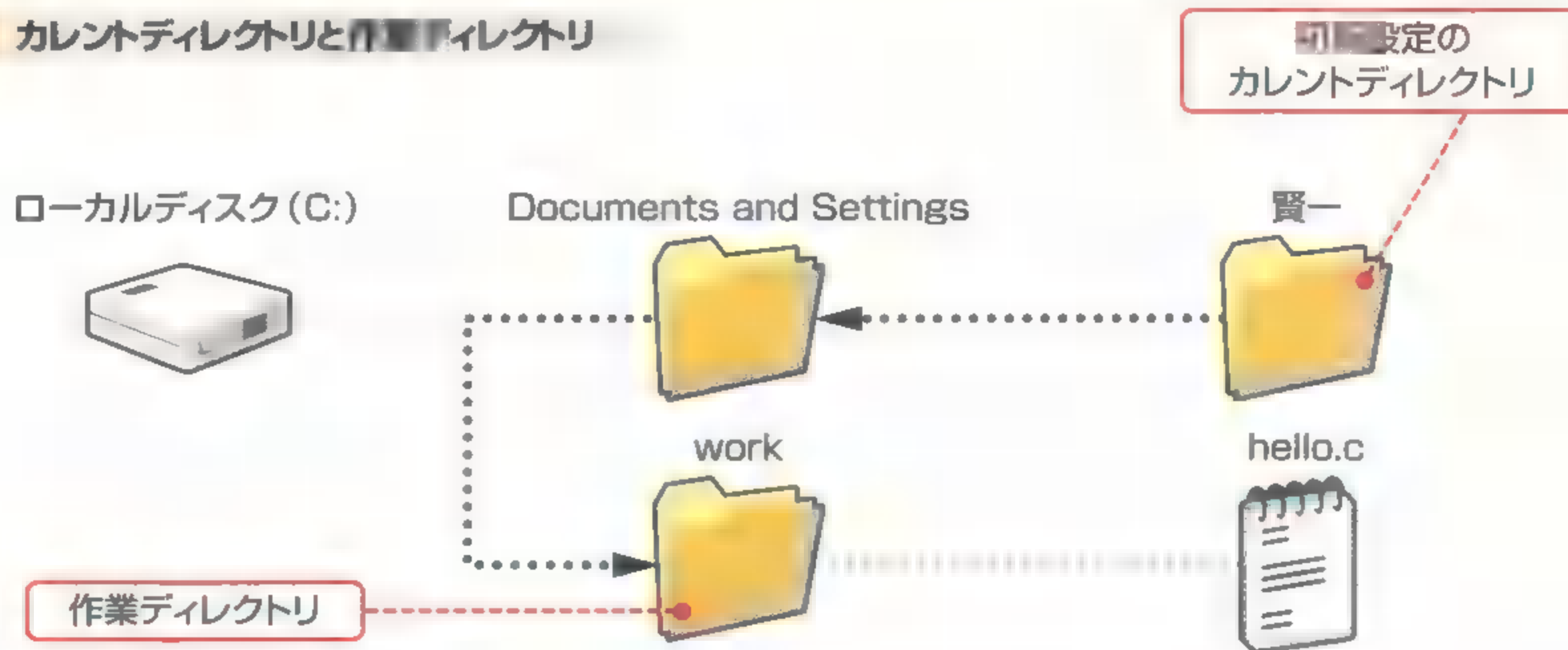
(4) カーソル

文字列の入力位置を示します。

4. コマンドの入力

■ カレントディレクトリの移動

初期設定のままでは、コマンドプロンプトを起動した直後にはカレントディレクトリが、「**C:¥Documents and Settings¥(ユーザー名)**」になります。そのため、ソースファイルをコンパイルするには、まずカレントディレクトリを作業ディレクトリ(本書では「**C:¥work**」)に移動する必要があります。

図5　カレントディレクトリと作業ディレクトリ

カレントディレクトリを移動するには、cdコマンドを利用します。

構文　CDコマンド（Change Directory）

```
cd 移動先のパス
```

「移動先のパス」は絶対パスと相対パスのどちらによっても指定することができます。なお、すべてのコマンドは[Enter]を押すと実行されます。

カレントディレクトリを作業ディレクトリ（C:¥work）に移動するには、コマンドプロンプトで「cd c:¥work」と入力し、[Enter]を押します。

なお、相対パスを使い、「cd ..¥..¥work」と入力してもかまいません。

■ カレントディレクトリの内容の表示

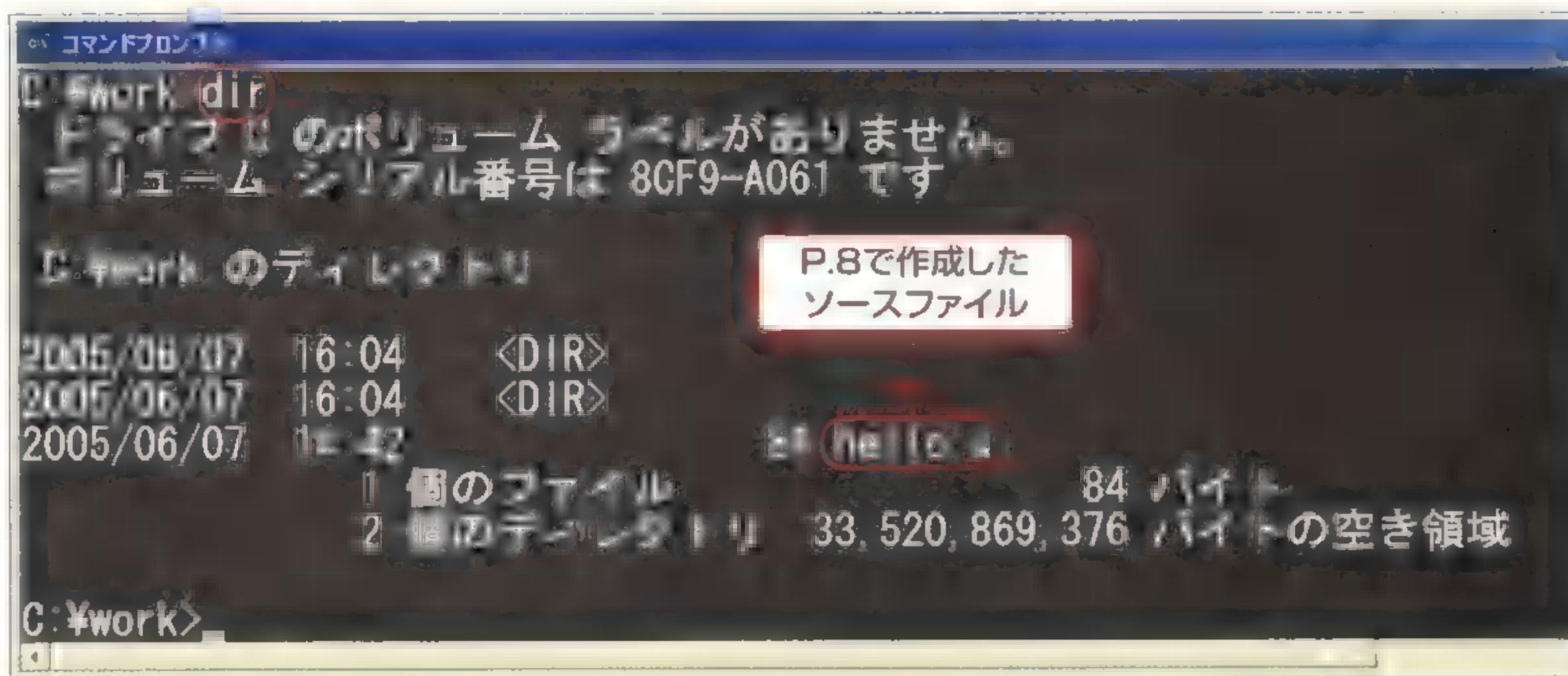
ディレクトリの内容を表示するには、dirコマンドを利用します。

構文　DIRコマンド

```
dir ディレクトリのパス
```

「ディレクトリのパス」は、絶対パスと相対パスのどちらでも指定することができます。なお、「ディレクトリのパス」の指定を省略すると、カレントディレクトリの内容が表示されます。

コマンドプロンプトで「**dir**」と入力し、Enterを押すと、カレントディレクトリの内容が一覧で表示され、P.8で作成したソースファイル「hello.c」が保存されていることを確認できます。「hello.c」は、次項のコンパイルの手順で必要になるため、あらかじめP.8の手順に従って作成してください。

```
C:¥work>dir
 ドライブ C のボリューム ラベルがありません。
 ボリューム シリアル番号は 8CF9-A061 です

 C:¥work のディレクトリ

2005/06/07  16:04    <DIR>          .
2005/06/07  16:04    <DIR>          ..
2005/06/07  1[illegible]:42                84 hello.c
               1 個のファイル                  84 バイト
               2 個のディレクトリ  33,520,869,376 バイトの空き領域

C:¥work>
```

P.8で作成したソースファイル

5. コンパイルと実行

■ ソースファイルのコンパイル

では、実際にソースファイルをコンパイルしてみましょう。本書では、**コンパイラ**として無償で提供されている**Borland C++ Compiler 5.5**を利用します（インストール方法についてはP.228を参照してください）。このコンパイラを使ってソースファイルをコンパイルするには、**bcc32**コマンドを用います。

構文　bcc32コマンド

```
bcc32 ソースファイル名
```

P.8で作成したソースファイル（hello.c）をコンパイルするには、「bcc32 hello.c」と入力してEnterを押します。

コンパイルが成功すると、コンパイラがいくつかのメッセージを表示し、再びプロンプトが表示されます。コンパイルが失敗した場合は「エラーメッセージ」が表示されます。エラーメッセージとして「'bcc32' は、内部コマンドまたは外部コマンド、操作可能なプログラムまたはバッチファイルとして認識されません。」と表示された場合は、**PATH環境変数**の設定が正しく行われていない可能性があります。付録1を参考にPATH環境変数の設定を確認してください。その他のエラーメッセージが表示された場合は、Sec.5を参考にソースコードの間違いを修正してコンパイルを成功させてください。

コンパイルが成功したら、dirコマンドでカレントディレクトリの内容を確認してみましょう。ソースファイル名の拡張子「.c」を「.exe」に変えたファイル「hello.exe」が新しく作成されていることを確認できます。これがコンパイルによって作成されたプログラム（**実行可能ファイル**）です。なお、.cや.exe以外にもオブジェクトファイルなど、プログラムに必要なファイルが作成されますが、これらはそのままにしておいてかまいません。

■ プログラムの実行

コンパイルしたプログラムを実行するには、プログラム（実行可能）ファイルの名前を入力します。入力の際、プログラムファイルの拡張子（.exe）は省略することができます。

構文　プログラム実行コマンド

```
実行可能ファイル名
```

たとえば「hello.exe」を実行する場合は、「hello.exe」もしくは「hello」と入力して[Enter]を押します。このとおり入力してみると、画面に「Hello! C world!」と表示され、プログラムが実行されたことを確認できます。

```
コマンドプロンプト
C:¥work>hello
Hello! C world!
C:¥work>_
```

Section 5

コーディングの注意点とよくあるエラー

覚えておきたいキーワード
- コンパイルエラー
- 警告

プログラミングは、大きく分けると「コーディング」と「コンパイル・実行」の2つの作業からなります。本セクションでは、これらの作業における注意点を解説します。特にコンパイルエラーは、慣れないうちは意味が読み取れないことがあるので、具体例をあげて解説します。

1. コンパイル時に発生するエラー

「**コンパイルエラー**」とは、コーディングの誤りをコンパイラが検出し、表示することをいいます。ソースコードにエラーがあると、コンパイラはエラーが発生した場所（行番号）や内容、エラーの個数などを知らせる「**エラーメッセージ**」を表示します。これを参考にプログラマはソースコードを修正します。

ただし、C言語のコンパイラが表示するエラーは、やや的外れな場合があります。簡単にいうと「嘘のエラーメッセージ」が表示されることがあります。

たとえば、次のようなソースコードをコンパイルしてみます。

Sample0103.c　コンパイルエラーが発生した行の特定

```
01  #include <stdio.h>
02
03  int main()
04  {
05      printf("Hello! C world!"),
06      return 0;
07  }
```

「;（セミコロン）」ではなく、「,（コロン）」が記述されています。

前のセクションで説明した手順に従い、コマンドプロンプトを開きます。「bcc32 Sample0103.c」と入力しEnterを押すと、コンパイラがエラーメッセージを表示して途中で終了します。

エラーメッセージは、次のように表示されます。これを見ると6行目に式の構文の間違いがあるように思われます。

```
C:¥work>bcc32 sample0103.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
sample0103.c:
エラー E2188 sample0103.c 6: 式の構文エラー(関数 main )
警告 W8070 sample0103.c 7: 関数は値を返すべき(関数 main )
*** 1 errors in Compile ***

C:¥work>
```

ソースファイルの6行目に
エラーがあるという
メッセージが表示されます。

実際には6行目自体には間違いはないのですが、そこが間違っているかのように表示されてしまいます。本当は5行目の関数の後に「;(セミコロン)」ではなく「:(コロン)」を記述してしまったことがエラーの原因です。

こういった行番号のズレは、「C言語のコンパイラだからしかたない」とあきらめてください。どのコンパイラも、同じようにおかしなエラーメッセージを出すことがあります。これは、C言語の書式の自由度が高く、確実にエラーだと断言できる場所がどうしてもズレてしまうためです。

ほとんどの場合において、エラーがあると表示された行の直前など、少し手前に本当の間違いがありますので、探してみてください。

■ 主なコンパイルエラー

スペルミス

```
itn main()
{
    printf("Hello! C world!");
    return 0;
}
```

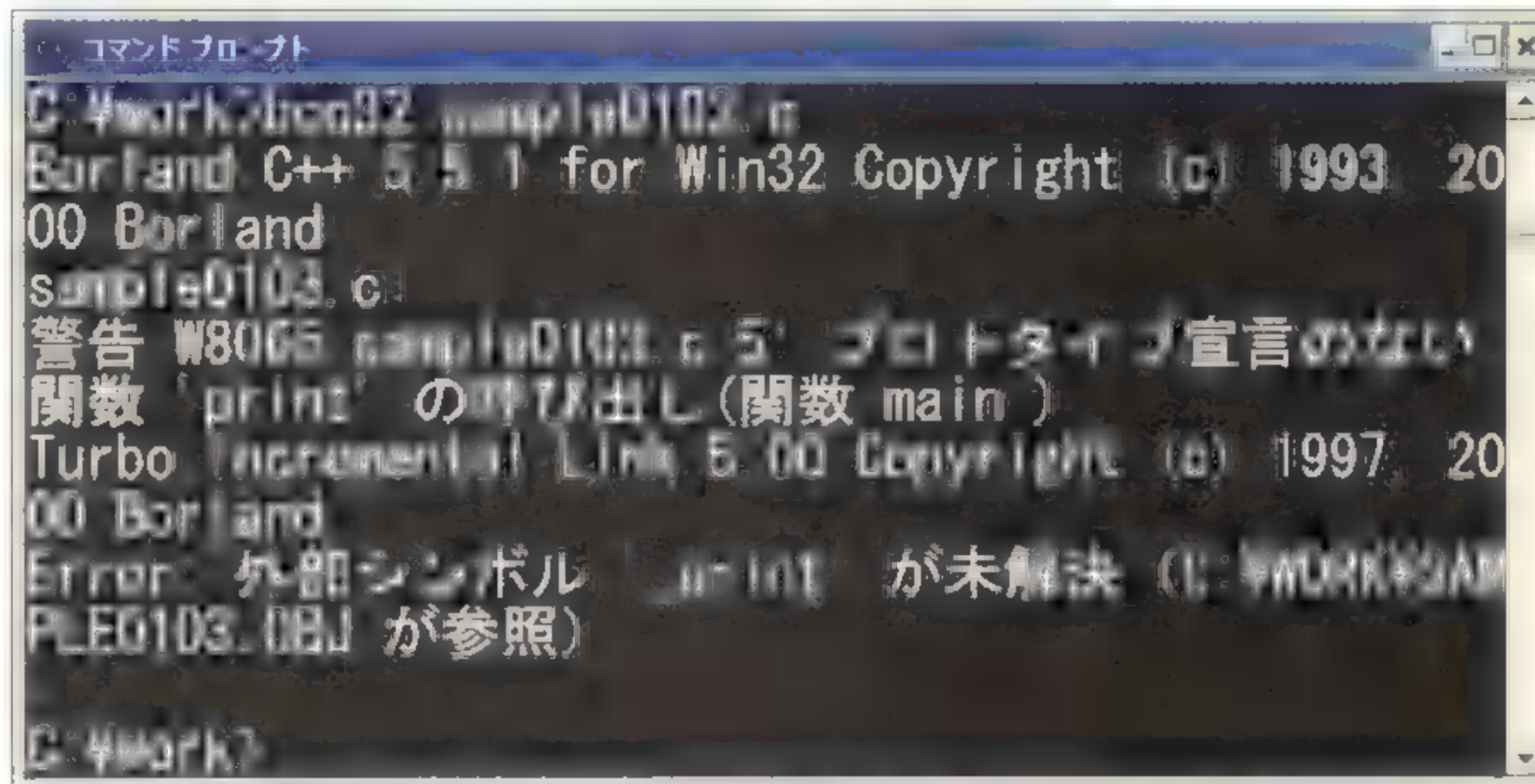
「int」と書くべきところに
「itn」と書いています。

```
C:¥work>bcc32 sample0103.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
sample0103.c:
エラー E2141 sample0103.c 3: 宣言の構文エラー
*** 1 errors in Compile ***

C:¥work>
```

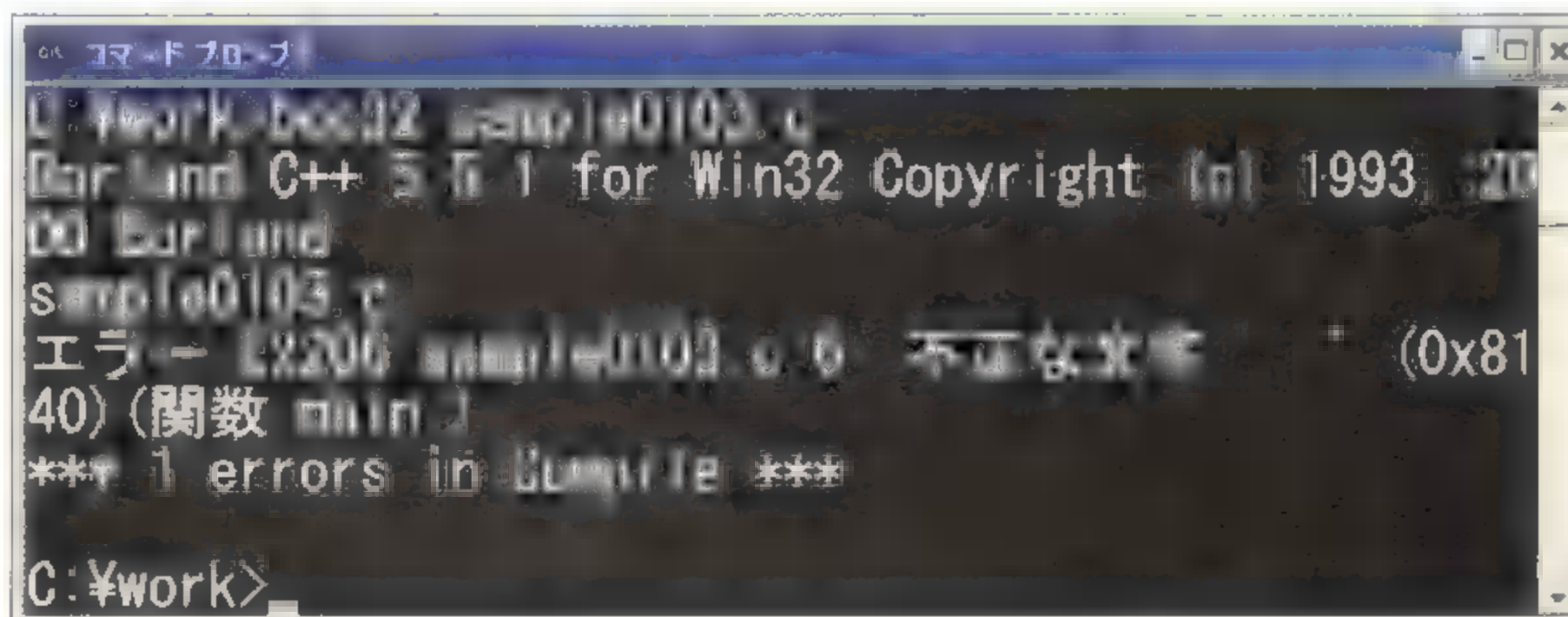
スペルミス（関数名）

```
int main()
{
    print("Hello! C world!");
    return 0;
}
```

「printf」と書くべきところに「print」と書いています。

```
C:¥work>bcc32 sample0103.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 20
00 Borland
sample0103.c:
警告 W8065 sample0103.c 5: プロトタイプ宣言のない
関数 'print' の呼び出し(関数 main )
Turbo Incremental Link 5.00 Copyright (c) 1997, 20
00 Borland
Error: 外部シンボル '_print' が未解決 (C:¥WORK¥SAM
PLE0103.OBJ が参照)

C:¥work>
```

全角スペースの混在

```
int main()
{
    printf("Hello! C world!");
　  return 0;
}
```

ソースコード中に全角スペースを使うことはできません。

ブロックの閉じ忘れ

```
int main()
{
    printf("Hello! C world!");
    return 0;
```

「{」で開いたブロックは、必ず「}」で閉じる必要があります。

```
コマンド プロンプト
C:¥work>bcc32 sample0103.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
sample0103.c:
エラー E2134 sample0103.c 8: 複合文に } がない(関数 main )
*** 1 errors in Compile ***

C:¥work>
```

Borland C++ Compiler 5.5では、上のエラーメッセージが表示されますが、コンパイラによっては「予期せぬEOF」というメッセージが表示される場合があります。「EOF」とはEnd Of Fileの略で、ファイル終端を意味します。つまり「関数のブロックが閉じられる前にEOFがきた」ということなので、上に示すエラーと同じ意味です。

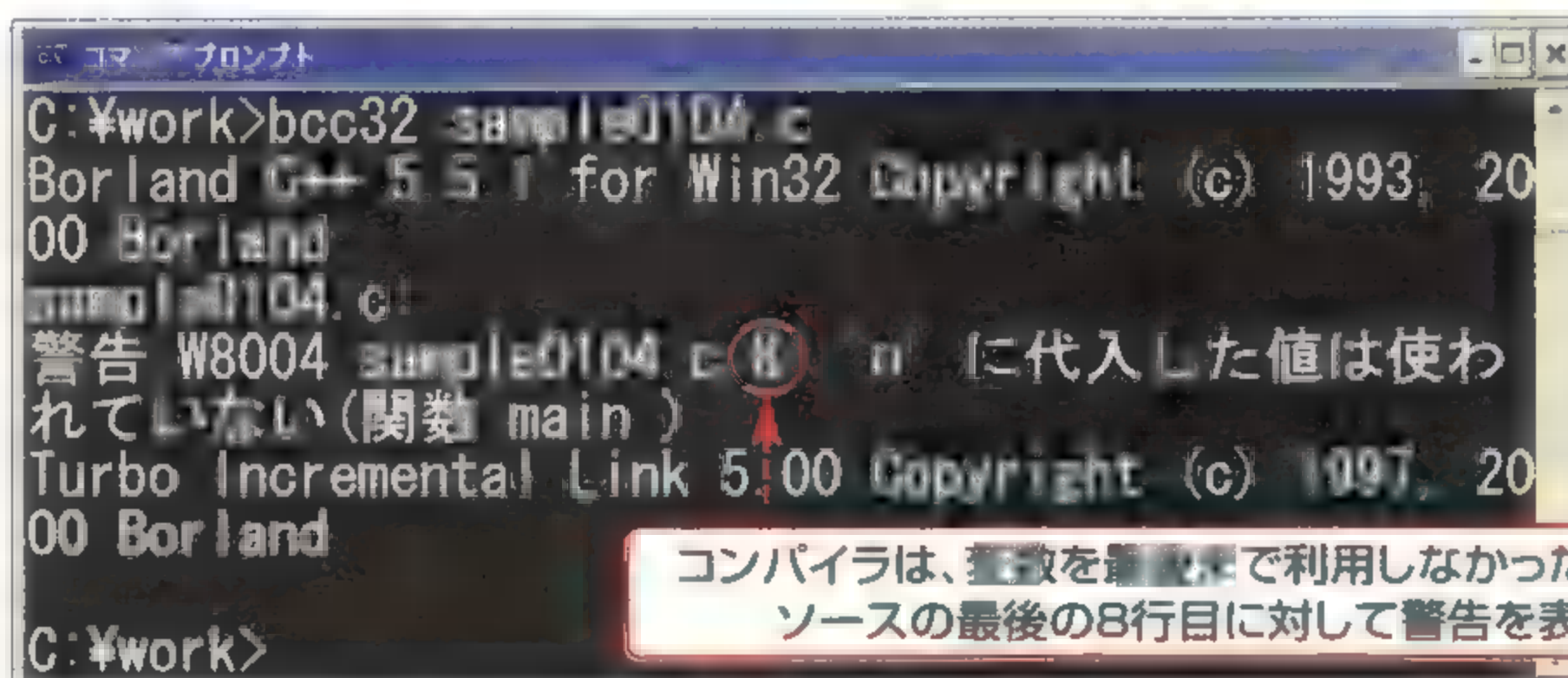
■ 警告とは…

「**警告**」とは、コンパイルエラーではないがプログラマが誤ってソースコードを記述している可能性がある部分について、コンパイラが「ここが間違っている可能性がある」と表示することをいいます。

C言語の文法は自由度が高いため、ソースコードを間違って記述していても、文法的には誤りではないという場合が比較的多くあります。このような場合に、コンパイラは警告を表示してプログラマに伝えてくれます。

ただし、警告は間違っている「可能性がある」ことを表示するだけですので、警告が表示されてもソースコードに問題がない場合もあります。

たとえば、C言語ではさまざまなデータを格納するために「変数（P.28参照）」を使いますが、ソースコードの中で「変数を使う」と宣言しておきながら、最後まで利用しなかった場合などに、コンパイラは警告を表示します。

警告が表示される例を次に示します。

Sample0104.c　警告メッセージの表示（変数の未使用）

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int n = 0;
06      printf("Hello! C world!"),
07      return 0;
08  }
```

ここで「[illegible]nを利用する」と宣言しています。

プログラムの最後まで変数nは利用されません。

コンパイラは、[illegible]で利用しなかったという意味で、ソースの最後の8行目に対して警告を表示します。

Column

P.2のSec.1で紹介したとおり、C言語には[illegible]標準規格があります。C言語の規格は[illegible]標準化[illegible]構（ISO、International Organization for Standardization）によって、制定されました。

実は、C言語の標準規格には大きく分けて次の3つがあります。

(1) C90

　1990年に、ISOによって制定された最初の規格です。

(2) C95

　1995年に、主にワイド文字に関するライブラリの追加などが行われた規格です。C90に対する補足として制定されました。

(3) C99

　1999年に制定された、「C言語の第2版」と呼ばれる規格です。新しい予約語が追加されたり、標準で複素数がサポートされたりして、大幅に仕様が拡張されました。

C99は規格がまだ新しく、追加された要素も専門的であることから、本書ではC99特有の事柄については説明を割愛します。また、本書ではこれらのどの規格でも共通するC言語の基本的な機能について解説します。新しい規格では使えなくなる、というものはありません。

Column WinMainが未解決?

もしこの章のプログラムをコンパイルしようとしたときに「外部シンボル'WinMain'が未解決」というエラーが表示されたのなら、プログラミングツールの使い方を調べてから作業を始める真面目なプログラマの素質を持っているといえるでしょう(実は、コンパイラの使い方を調べないと、このエラーはなかなか出せません)。
このエラーは、この章で用いたプログラム例を「Windows GUIアプリケーション」としてコンパイルしようとした場合に発生します。

構文 GUIアプリケーションのコンパイル

```
bcc32 -W ソースファイル名
```

「-W」を付けます。

GUIアプリケーションを作成したい場合、通常のC言語プログラミングとは異なったルールで記述しなければいけない部分も多く、また概念もプログラミングの入門者にとっては難解で複雑です。そのため、本書ではGUIアプリケーションのプログラミングについては触れません。
参考までに、もっとも単純なGUIアプリケーションのプログラムを次に示します。前述の方法でコンパイルして実行すると、画面にダイアログボックスが表示されます。興味があれば試してみてください。

Sample0105.c 単純なGUIアプリケーション

```
01 #include <windows.h>
02 int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPre,
03                    LPSTR lpCmdLine, int nCmdShow)
04 {
05     MessageBox(0, "Hello! C world!", "Welcome", MB_OK);
06     return 0;
07 }
```

まとめ　第1章：初めてのCプログラミング

この章では、C言語の概要と特徴について解説しました。また、C言語でプログラムを作成するしくみや、その開発環境についても簡単に触れました。C言語でプログラムを作成する際の最低限の約束事や、よくある間違いについてはきちんと覚えておきましょう。

第1章で学習したこと

- C言語は、世界でもっとも使われているプログラミング言語のひとつである。
- C言語のプログラムは「ソースコード」を作成し、それを「コンパイル」することで作成することができる。
- コンパイルを行うソフトウェアのことを「コンパイラ」という。
- 世界標準規格として「ANSI C」が規定されており、ANSI Cに従ったプログラムであれば、開発環境が変わってもそのままソースコードを利用できる。
- C言語のプログラムは、必ず「main()関数」から処理が始まる。
- 画面に文字を表示するためには「printf()関数」を利用する。
- ソースコードに誤りがあってコンパイルが失敗することを「コンパイルエラー」という。
- コンパイラのエラーメッセージは、ときどき頼りにならないことがあるので、エラーが発生した行の周辺もよく確認する。

ステップアップ！

C言語では、さまざまなプログラムを作成することができますが、本書ではCUIアプリケーションを作成することを通して、プログラミングの基礎を学習します。第2章以降では、実際にプログラムの基礎知識の解説を始めます。解説内容を読むだけではなく、サンプルプログラムを動作させてみたり、自分でプログラムを作成してみたりして、理解を深めてください。それがプログラミング上達の近道となります。

第2章

Visual Learning Introduction of C

変数と演算子

Section 6

覚えておきたいキーワード
- 変数
- データ型
- 変数の宣言

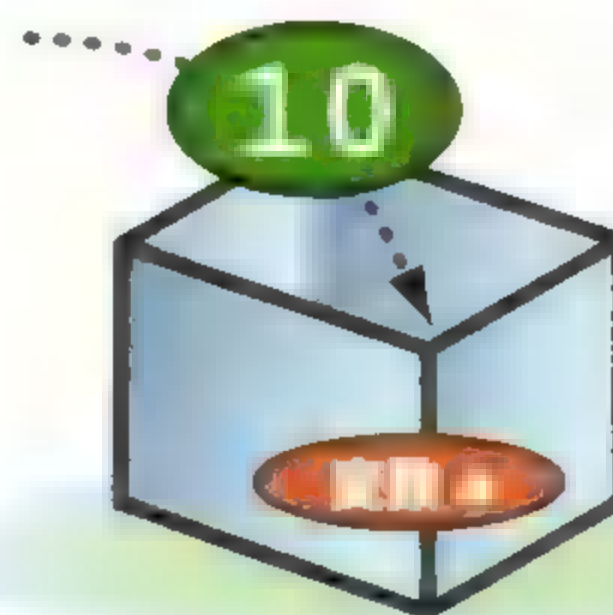
変数

プログラムがさまざまなデータを処理するには、データを一時的に記憶しておく「場所」が必要です。C言語では、変数を利用して、データを記憶しておく場所を作成します。変数とは、いわばデータを入れておく「箱」のようなものです。

1. 変数の定義

■ 変数とは…

プログラムでデータを扱って計算などを行いたい場合、一時的にでもデータを記憶しておきたいことがあります。このデータを一時的に保存しておく場所のことを「**変数**」といいます。変数とは、いわばデータを入れておく「箱」のようなものです。中に入れるデータは、後から自由に変更することができます。

図1　変数

■ コンピュータの数値表現方法

変数の大まかな意味は「データを入れる箱」ということです。では、具体的にコンピュータはどうやって数値を記憶するのでしょうか。

まず、コンピュータが数値を表現する方法を説明します。よく「デジタルの世界は0と1しかない」という言葉を耳にします。これは本当のことです。コンピュータは「**0**」と「**1**」の2つを組み合わせて数値を表現します。

たとえば、1個の0または1では2通りの情報しか表現できませんが、2個の0または1を用いると、2の2乗で4通りの情報が表現できるようになります。この1個の0または1のことを「**ビット**」と呼びます。また、ビットが8個集まったものを「**バイト**」と呼びます。

図2　ビット

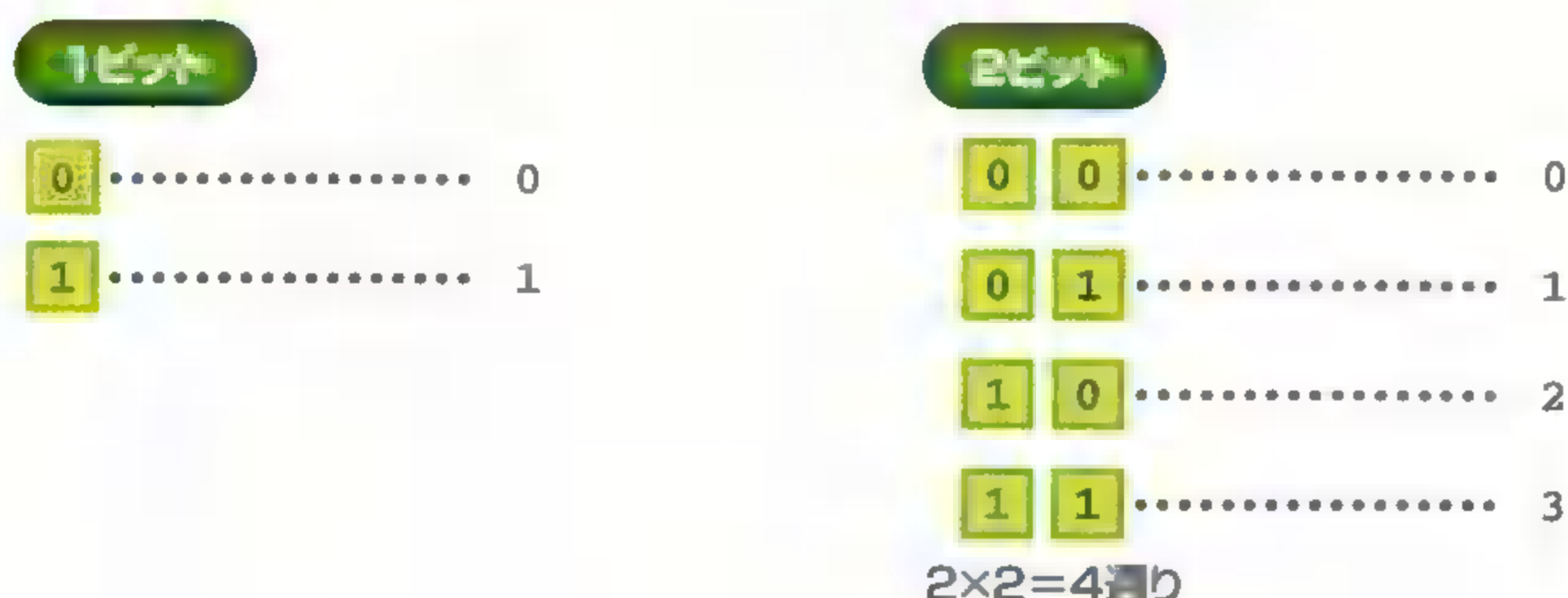
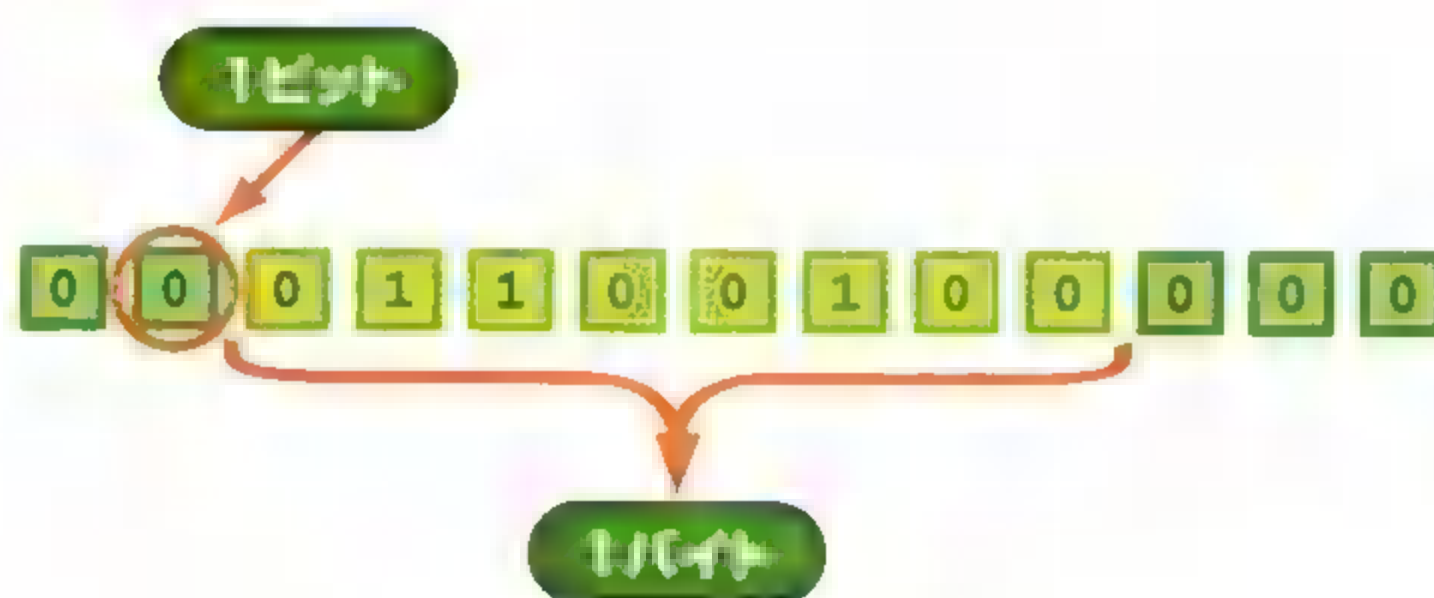



図3　ビットとバイト

1バイトは8ビットであるため、2の8乗で256通り、2バイトは16ビットであるため、2の16乗で65,535通りの情報が表現できるようになります。

nビットで表現できる情報＝2^n通り

コンピュータは、ビットのさまざまなパターンにより数値を表現します。

■ コンピュータの文字表現方法

文字の表現も、基本的には数値の表現と同じです。違うのは「コンピュータ世界共通の文字の一覧表（ASCIIコード）」があり、あるビットパターンをある文字に対応させて表現していることだけです。

たとえば、数値「65」は文字「A」であると決められています。この場合、数値「65」のことを「文字コード」といいます。文字コード一覧については付録2を参照してください。

コンピュータから見ると文字コードの65（つまり文字「A」）なのか、本当の数値の65なのかを区別できません。従って、プログラマがコーディングの際に文字または数値のどちらであるか、「この値は文字として扱うように」といった指示を行う必要があります。

2. 変数の型

C言語では変数を使って数値や文字を記憶しますが、扱うデータの種類に応じて「**データ型**」が用意されています。また、データ型ごとに扱えるデータの種類とバイト数が決められています。

C言語で利用できるデータ型には、次のようなものがあります。

表1　変数の型

種類	データ型名	サイズ	扱える数値の範囲
文字型	char	1バイト	英数字1文字(−128〜127)
	unsigned char	1バイト	0〜255
整数型	short	2バイト	−32,768〜32,767
	int	4バイト	−2,147,483,648〜2,147,483,647
	long	4バイト	−2,147,483,648〜2,147,483,647
符号なし整数型	unsigned short	2バイト	0〜65,535
	unsigned int	4バイト	0〜4,294,967,295
	unsigned long	4バイト	0〜4,294,967,295
浮動小数点数型	float	4バイト	$1.18 \times 10^{-38} \sim 3.40 \times 10^{38}$
	double	8バイト	$2.23 \times 10^{-308} \sim 1.79 \times 10^{308}$

ただし、**int**や浮動小数点数型はコンパイラを稼働させる環境によってバイト数が異なる場合があります。この表ではWindows XP上でコンパイラとしてBorland C++ Compiler 5.5を使う場合のデータ型を記載しています。

3. 変数の宣言

■ 変数宣言の構文

実際にC言語で変数を利用するにはまず変数を**宣言**しなければなりません。変数を宣言するには、次のようにソースコードを記述します。

構文　変数宣言

```
データ型　変数名;
```

たとえば、int型の変数aを宣言したい場合は次のように記述します。

```
int a;
```

変数宣言は命令文のひとつです。文の最後に「;（セミコロン）」を忘れずに記述しましょう。

■ 識別子のルールと予約語

変数などに付ける名前のことを「**識別子**」といいます。先ほどの例でいうと「a」が識別子です。識別子を利用する際は、以下に示すルールに従います。

(1) 半角英数字（a～z、A～Z、0～9）、アンダースコア（_）のいずれかを利用する。ただし識別子の先頭に数字は使用できない。

(2) 大文字と小文字は別のものとして区別される。

(3) 次に示す「C言語の予約語」は識別子として使用できない。

表2　C言語の予約語

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

値の代入

変数を宣言して「箱」を用意したら、次にその箱に数値という「もの」を入れてみましょう。変数には文字型や整数型などさまざまなデータ型がありますが、それぞれに応じた代入のしかたがあります。このセクションでは、データ型に応じた代入の方法を解説します。

1. 変数の利用

■ 代入とは...

宣言した変数を利用するために、変数に値を記憶させます。この変数に値を記憶させることを「**値を代入する**」といいます。変数に値を代入するには、「**=**」を利用して次のように記述します。「**=**」のことを「代入記号」といいます。

数学などでは「=の左側と右側の値は等しい」という意味で使われますが、C言語ではまったく違う意味で使われるため、注意しましょう。

構文　値の代入

```
変数名 = 値;
```

■ 整数の代入

変数のデータ型によって、代入可能な値およびその記述方法が異なります。まず、整数型の場合を見てみましょう。

たとえば、**a**という名前の**int**型の変数に「10」という整数を代入するには、次のように記述します。

```
int a;
   代入
a = 10;
```

C言語では、代入記号の左側にあるものに、代入記号の右側にあるものをコピーします。

また、変数に代入した値を別の変数に代入することも可能です。たとえば、

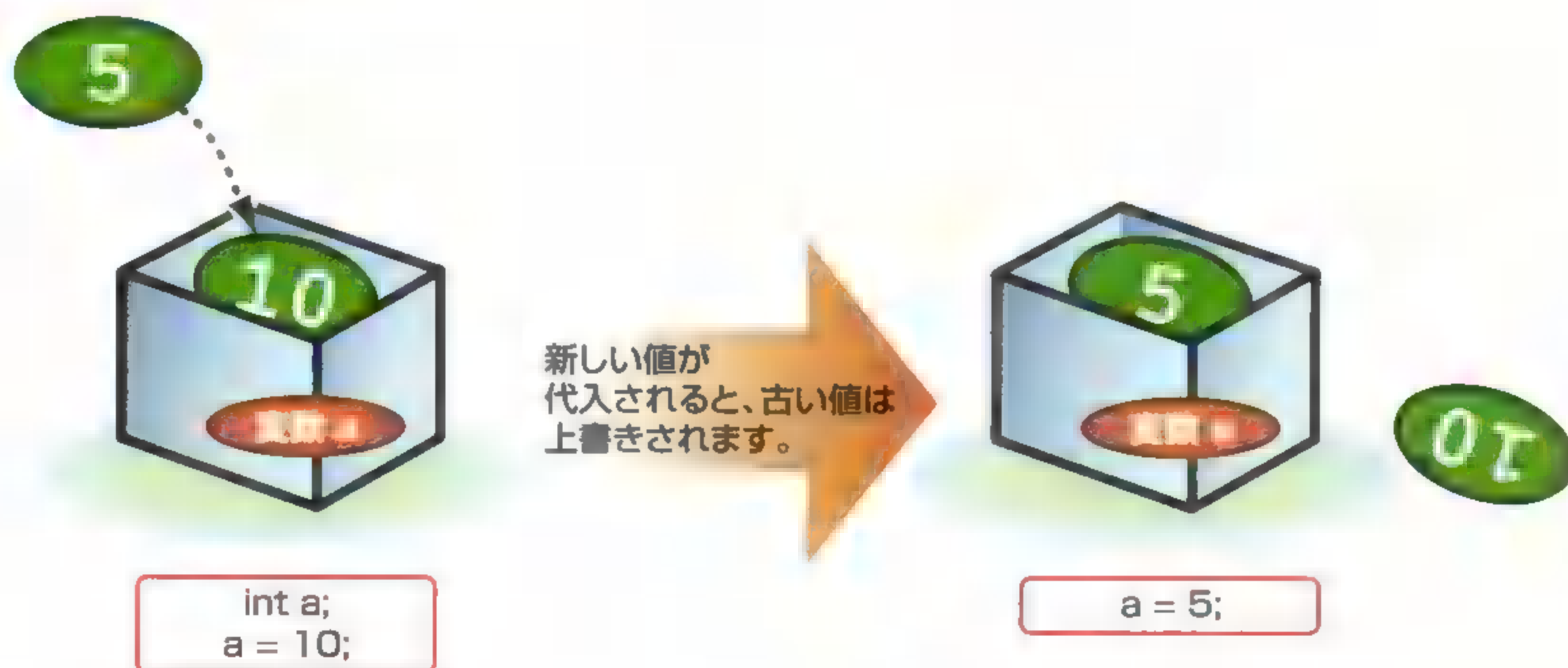
```
int a;
int b;
a = 10;
 代入
b = a;
```

と記述すると、変数aにまず「10」が代入され、次に変数bに変数aの中身の「10」が代入されます。

■ 変数は1つの値しか覚えられない

変数は、1つの値しか覚えることができません。従って、次に示すように何か値が代入されている変数に違う数値を代入すると、もともと記憶していた値は新しく代入された値によって上書きされてしまいます。

図1　新しい値の代入

5
10
新しい値が
代入されると、古い値は
上書きされます。
5
10
int a;
a = 10;
a = 5;

■ 8進数と16進数の代入

C言語では、変数に整数を代入する場合、代入する値を10進数だけでなく8進数や16進数で記述することができます。

10進数とは「数値を0～9で表現し、9に1を加えると10になる」（桁が上がる）という数値の

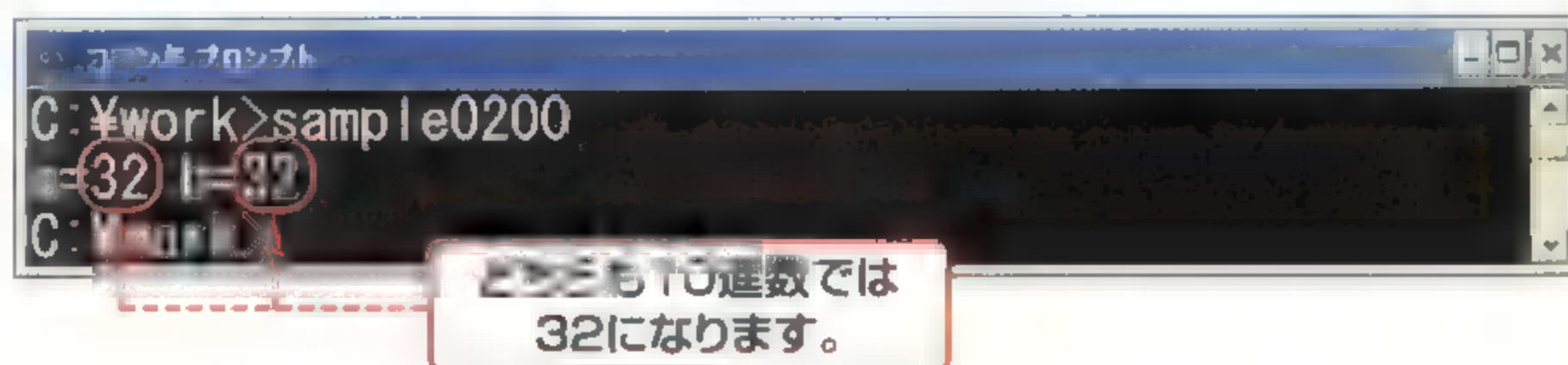
表現方法のことです。8進数は同様に「数値を0〜7で表現し、7に1を加えると10になる」、16進数は「数値を0〜9とA、B、C、D、E、Fで表現し、Fに1を加えると10になる」という数値の表現方法を意味します。

C言語では、変数に8進数で表記した整数を代入したい場合は、値の先頭に「**0**」を付けて記述します。また、16進数で表記した整数を代入したい場合は、値の先頭に「**0x**」を付けて記述します。なお、2進数の値はC言語では表現できません。

表1　8進数と16進数の記述

表記	記述	10進数で表現した場合の値
8進数	010	8
16進数	0x10	16

これらを利用したプログラムは、次のようになります。なお、**printf()**関数は画面に値を表示する関数です。詳細については、P.36を参照してください。

```
int a = 040;  /* 8進数で40(10進数で32)を代入 */
int b = 0x20; /* 16進数で20(10進数で32)を代入 */
printf("a=%d b=%d", a, b);
```

■ 浮動小数点数の代入

浮動小数点数とは、簡単にいうと「小数点を含む値を扱うための型」です。浮動小数点数型の値を代入するには、**float**型、もしくは**double**型の変数を利用します。

```
float c = 10.2;
double d = 11.3;
```

なお、**int**など整数型の変数に小数点を含む値を代入しようとすると、小数点以下が切り捨てられます。

■ 文字の代入

文字を代入する場合、**char**型もしくは**unsigned char**型の変数を利用し、代入したい文字を「'(シングルクォーテーション)」で囲んで記述します。

```
char e = 'A';
```

代入したい文字を、「'」で囲みます。

文字は「**文字コード**」を用いて表現されます。そしてC言語では、1つの文字を「'」で囲むと、その文字の文字コードを取得できます。前述の例では、「**char**型の変数**e**にAという文字を表す文字コードを代入する」という意味になります。

なお、この方法で代入できるのは1つの文字だけです。複数の文字(つまり文字列)を記憶する方法についてはSec.14で解説します。

また、漢字などの2バイト文字も、**char**型変数に代入することはできません。漢字を代入したい場合は、たとえ1文字でも文字列として扱います。

■ 変数の初期化

変数を扱う際には、ある問題に注意しなければなりません。変数を宣言したのに値を代入していない状態では、その変数の値は不定の値(つまりデタラメな値)になっています。この状態で変数を利用すると、エラーの原因になりかねません。

そこで、デタラメな値のままで変数を利用しないようにするために、あらかじめ変数に値を代入しておきます。これを「**初期化**」、そのときに代入する値のことを「**初期値**」といいます。

たとえば、次のようなプログラムの変数**a**の初期値は「10」となります。

```
int a;
a = 10;
```

変数aの初期値は10になります。

また、次に示すように、あらかじめ変数の初期値が決まっている場合などに、変数の宣言と初期化を同時に行うこともできます。

```
int a = 10;
```

Section 8

printf()関数による画面表示

覚えておきたいキーワード
- printf()関数
- 変換指定子
- エスケープシーケンス

画面に文字を表示するにはprintf()関数を利用します。今までは「Hello」といった文字列を表示させるだけでしたが、変数に代入した値や、改行、タブ文字なども表示することができます。このセクションではprintf()関数の使い方について解説します。

1. printf()関数の利用

■ 画面に文字を表示させる方法

前のセクションでは変数へ値を代入する方法を学習しました。では、変数に値が正しく代入されているかどうかを画面に表示して確認してみましょう。画面に変数を表示するためには、**printf()**関数を利用します。

関数の詳細についてはSec.16で説明します。ここでは、C言語のプログラムでもっとも利用する機会の多い**printf()**関数の利用方法を覚えておきましょう。

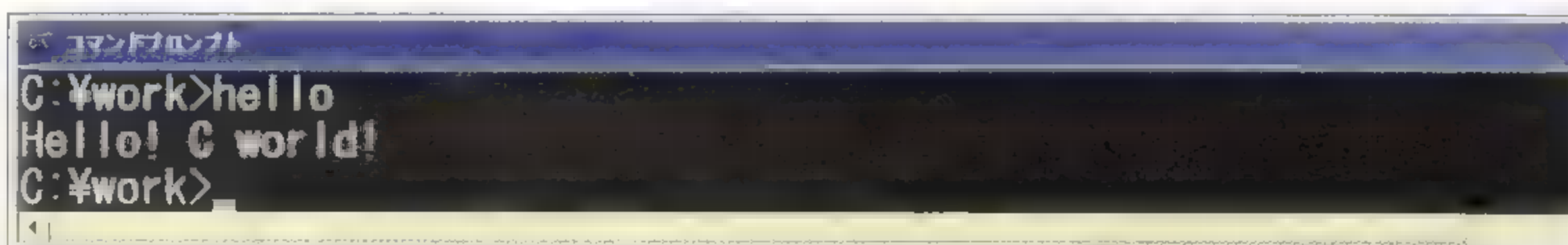
printf()関数で文字列を表示させるには、次のように記述します。

printfに続いて「**(**」、「**"**(ダブルクォーテーション)」で囲んだ表示文字列、「**)**」を記述し、最後に「**;**(セミコロン)」で閉じます。

たとえば、Sec.4で行ったように「Hello! C world!」と画面に表示させるには、次のように記述します。

```
printf("Hello! C world!");
```

```
C:¥work>hello
Hello! C world!
C:¥work>_
```

2 変数と演算子

■ 変換指定子

printf()関数で文字列を表示するのは非常に簡単ですが、変数に代入されている値を表示するにはちょっとした工夫が必要です。

変数を表示するためには、「"」で囲んだ文字列の中に「ここに変数を入れる」ことを示す特殊な記号を記述します。この特殊な記号のことを「**変換指定子**」といいます。

構文　変換指定子

```
printf("……変換指定子……", 変数);
```

変換指定子には表示したいデータ型によっていくつかの種類が用意されています。主な変換指定子には、次のようなものがあります。

表1　変換指定子

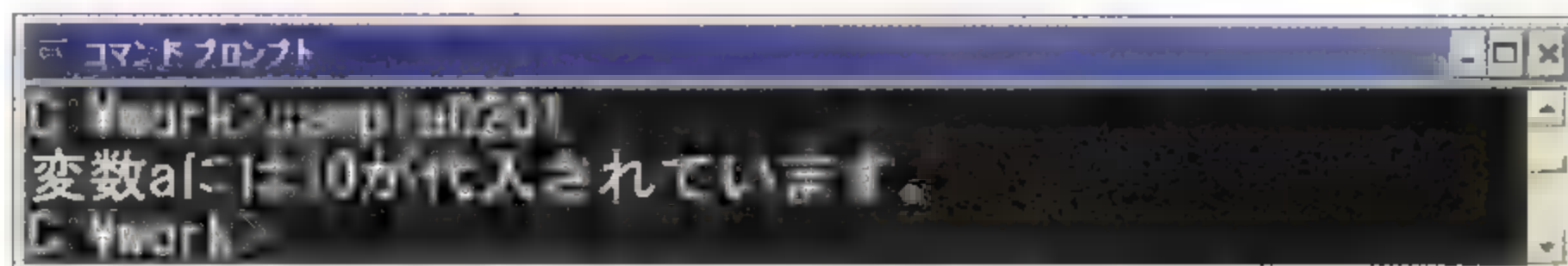
変換指定子	機　能
%d	整数型の値を表示します。
%f	浮動小数点数型の値を表示します。
%c	文字型の値を1文字表示します。
%s	文字列を表示します。文字列についてはSec.14参照。
%o	指定した値を8進数で表示します。
%x	指定した値を16進数で表示します。

変換指定子を利用したコードは、次のようになります。

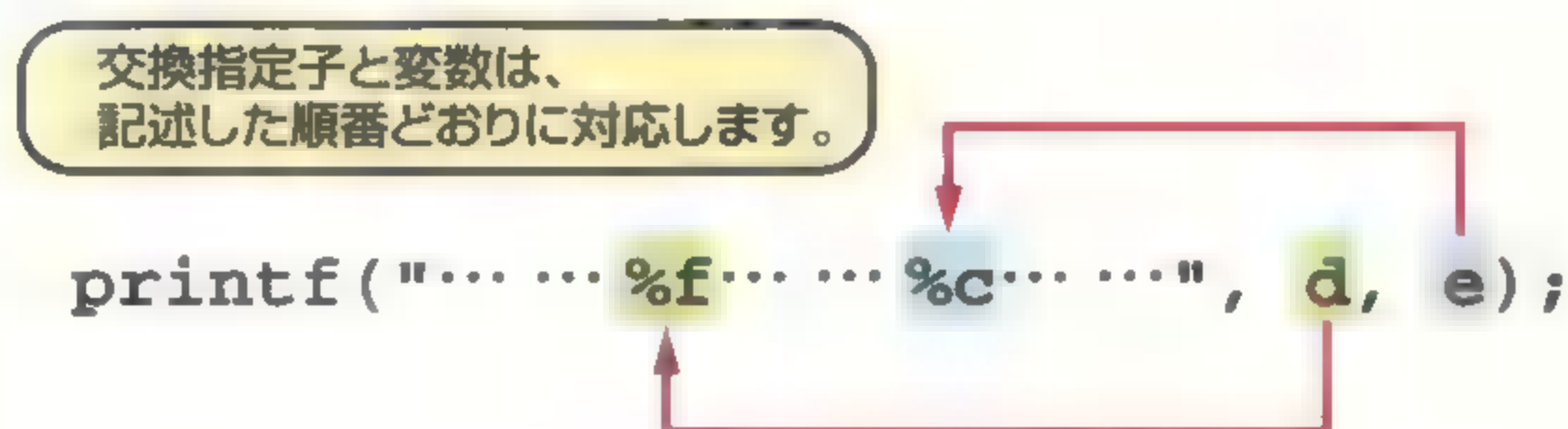
Sample0201.c　変換指定子の利用

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int a;
06      a = 10;
07      printf("変数aには%dが代入されています。", a);
08      return 0;
09  }
```

変数を表示するための変換指定子「%d」を利用しています。

これをコンパイルし実行すると、

のように、「**%d**」の位置に、**a**に代入した値が表示されます。

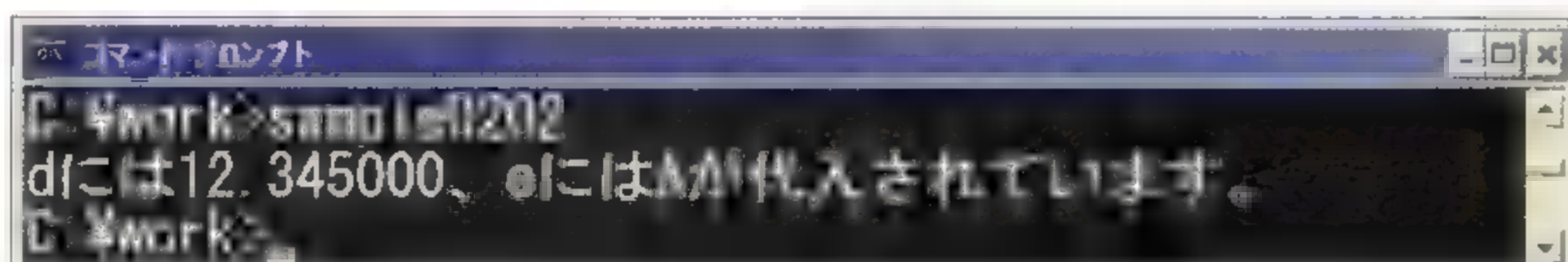
また、変換指定子と「,（カンマ）」の後に並べる変数を増やすことで、複数の変数を1つの**printf()**関数で表示させることもできます。

図1　変換指定子と変数

たとえば、浮動小数点数と文字を表示する場合は、次のように記述します。

Sample0202.c　複数の変換指定子の利用

```
01  #include <stdio.h>
02
03  int main()
04  {
05      double d;
06      char e;
07      d = 12.345;
08      e = 'A';
09      printf("dには%f、eには%cが代入されています。", d, e);
10      return 0;
```

最初に記述した変換指定子の位置には、1つ目に指定した変数の値が表示されます。

2つ目に記述した変換指定子の位置には、2つ目に指定した変数の値が表示されます。

■ エスケープシーケンス

printf()関数では、数値や文字だけでなく、**改行**記号などの特殊記号を扱うことができます。ただし、そのためには、「**エスケープシーケンス**」を利用しなければなりません。

例として、まず、変数を2つ表示する次のようなプログラムを作成してみましょう。

Sample0203.c　2回の画面表示

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int a = 10;
06      int b = 5;
07      printf("a は %d です。", a);
08      printf("b は %d です。", b);
09      return 0;
10  }
```

そしてこのプログラムを実行すると、次のような画面になります。

```
コマンド プロンプト
C:¥work>sample0203
a は 10 です。b は 5 です。
C:¥work>
```

この例では、変数**a**の表示と変数**b**の表示がつながってしまいます。**printf()**関数で文字列を2回表示しても、その間で画面上で改行が行われるわけではありません。

改行を行いたい場合には、変換指定子と同様に「ここに改行を入れる」ことを示す特殊な記号を記述します。改行をはじめとして特殊な記号にはいくつかの種類があります。このように特殊な記号であることを示す表記方法を「エスケープシーケンス」といいます。

エスケープシーケンスは「エスケープコード」+「制御文字」からなります。たとえば、改行を出力するためには「**¥n**」というエスケープシーケンスを使いますが、このうち「¥(円記号)」がエスケープコード、「**n**」が制御文字にあたります。

コンパイラは文字列中に「¥」が現れると、その次の文字を制御文字とみなし、エスケープコードと合わせてエスケープシーケンスとして解釈します。C言語のプログラミングにおいてもっともよく利用するエスケープシーケンスは改行を表す「¥n」です。

主なエスケープシーケンスには、次のようなものがあります。

表2 エスケープシーケンス

記　号	機　能
¥n	改行を行います。
¥t	タブを入れます。
¥¥	「¥」を表示します。
¥?	「?」を表示します。
¥0	NULL(ヌル)を表示します。
¥'	「'(シングルクォーテーション)」を表示します。
¥"	「"(ダブルクォーテーション)」を表示します。
¥ooo	8進数oooの文字コードの文字を表示します。
¥xhh	16進数hhの文字コードの文字を表示します。

エスケープシーケンスを利用すると、次のようなプログラムを書くことができます。

Sample0204.c 改行記号の利用

```
01 #include <stdio.h>
02
03 int main()
04 {
05     int a = 10;
06     int b = 5;
07     printf("a は %d です。¥n", a);
08     printf("b は %d です。", b);
09     return 0;
10 }
```

改行を表す「¥n」を利用しています。

最初の**printf()**関数の呼び出しにおいて「**¥n**」により改行を指定しているため、実行結果は次のようになります。

Column 変数を右揃えで表示したい場合

printf()関数で変数を表示した場合、左揃えに出力されます。これを右揃えに表示したい場合、変換指定子を次のように記述します。

```
int a = 10;
int b = 5;
printf("a=%4d¥n", a);
printf("b=%4d", b);
```

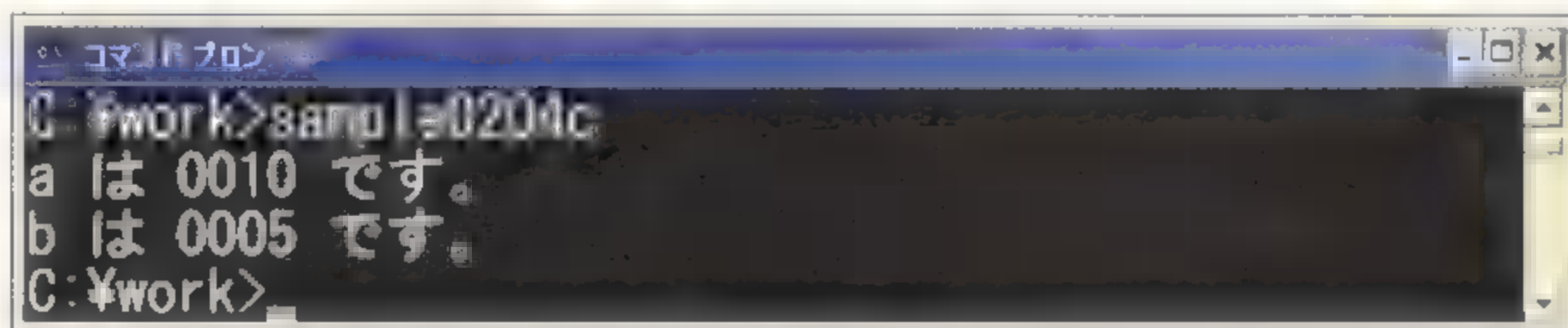
「%」と「d」の間に、表示する桁数を記述します。

これを実行すると、次のように表示されます。

printf()関数では、変換指定子の「%」と「d」の間に数字を記述して変数を表示する幅を指定することができます。設定した幅よりも変数の値の桁数が少ない場合は、余った桁に空白を表示します。設定した幅よりも値の桁数が多い場合は、通常の%dと同じ出力結果になります。

また、余った桁に空白の代わりに「0」を表示させるには、次のように記述します。

```
int a = 10;
int b = 5;
printf("a=%04d¥n", a);
printf("b=%04d", b);
```

先ほどと同様に「%」と「d」の間に数字を入れます。数字の先頭に「0」を付けると、余った桁が空白の代わりに「0」で埋められるようになります。このプログラムを実行すると、次のようになります。

演算子

覚えておきたいキーワード

- 算術演算子
- 代入演算子
- ビット演算子

C言語では、数値や変数を使ってコンピュータに演算を実行させます。これらの演算の組み合わせによりさまざまな処理を行い、プログラムを作り上げていくのです。演算を行う式は、演算の種類を表す演算子と演算対象の値から構成されます。

1. 式と演算子

■ 式とは…

プログラムは複数の演算を組み合わせて、さまざまな処理を行います。演算を行うには「**式**」を記述します。ほとんどの式は、数学における数式のように、演算の種類を表す「**演算子**」と演算の対象となる「**値**」から構成されます。式で利用する値のことを「**オペランド**」と呼びます。

たとえば3と7の足し算は、C言語で

```
3 + 7
```

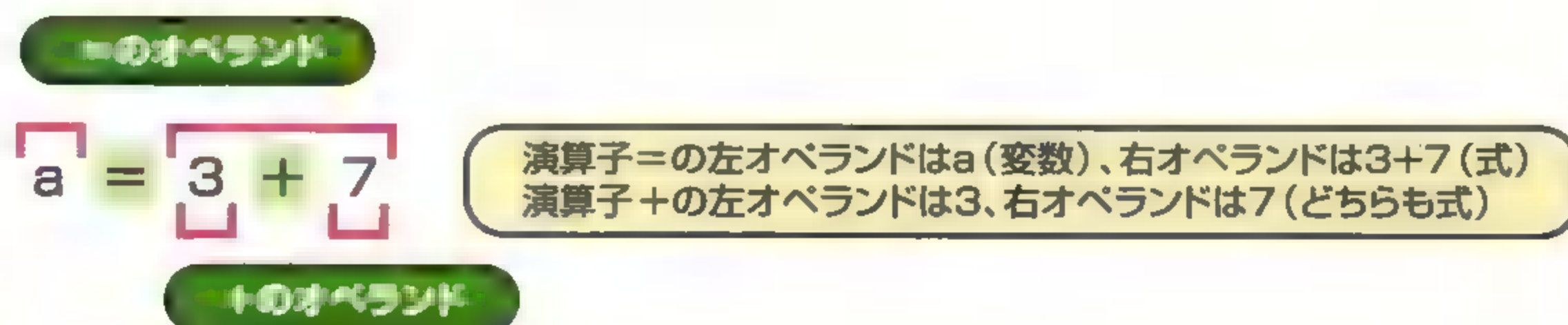
と記述します。ここでは「+」が演算子、「3」と「7」がオペランドにあたります。もちろん、計算した結果を変数に代入することも可能です。

```
int a = 3 + 7;
```

この場合、3と7を足した結果の「10」が変数aに代入されます。演算子は目的に応じてさまざまなものが用意されています。また、オペランドに変数や式を指定することもできます。

図1　演算子とオペランド

+のオペランド

演算子=の左オペランドはa（変数）、右オペランドは3+7（式）
演算子+の左オペランドは3、右オペランドは7（どちらも式）

■ 算術演算子

「**算術演算子**」は四則演算と剰余の計算を行うときに使用します。

算術演算子には、次のようなものがあります。なお、C言語では数値を0で割ると致命的なエラーになりますので十分に注意してください。

表1　算術演算子

演算子	使用例	意　味
+	a+b	aとbを足します。
−	a−b	aからbを引きます。
*	a*b	aとbを掛けます。
/	a/b	aをbで割った商を求めます。
%	a%b	aをbで割った余りを求めます。

算術演算子は必ず2つのオペランドをとります。このように演算対象のオペランドを2つとる演算子を「二項演算子」と呼びます。

■ 代入演算子

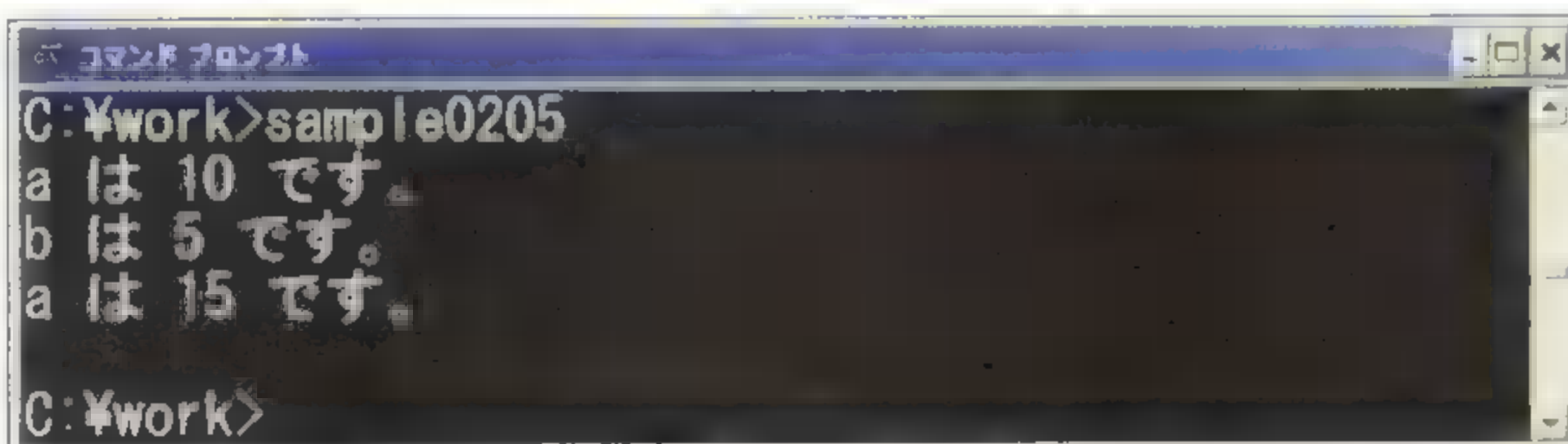
「**代入演算子**」とは、変数に値を代入するための演算子のことです。これまで変数に値を代入する際に利用してきた「**=**」も代入演算子のひとつです。この演算子は、「**=**」の右辺にある値または式の演算結果を左辺に代入する操作を行います。

代入演算子には次のようなものがあります。

表2　代入演算子

演算子	使用例	意　味
=	a=b	aにbを代入します。
+=	a+=b	aにa+bの結果を代入します。
−=	a−=b	aにa−bの結果を代入します。
*=	a*=b	aにa*bの結果を代入します。
/=	a/=b	aにa/bの結果を代入します。
%=	a%=b	aにa%bの結果を代入します。

このうち、算術演算の結果を代入するものを「**複合代入演算子**」と呼びます。表を見てわかるとおり、代入演算子も2つのオペランドをとる二項演算子です。

たとえば、「**a += 4**」は「**a = a + 4**」と同じ計算を意味するため、記述を簡略化することができます。なお、このような複合代入演算子を利用する場合、「**+**」などの演算子と「**=**」の間にスペースを入れてはいけません。

次に示すのは、算術演算子と代入演算子を利用したプログラムです。

Sample0205.c　演算子の利用

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int a = 10;
06      int b = a - 5;
07      printf("a は %d です。¥n", a);
08      printf("b は %d です。¥n", b);
09      a += b;
10      printf("a は %d です。¥n", a);
11      return 0;
12  }
```

a−5を計算し、その結果をbに代入しています。

aに、a+bの計算結果を代入しています。

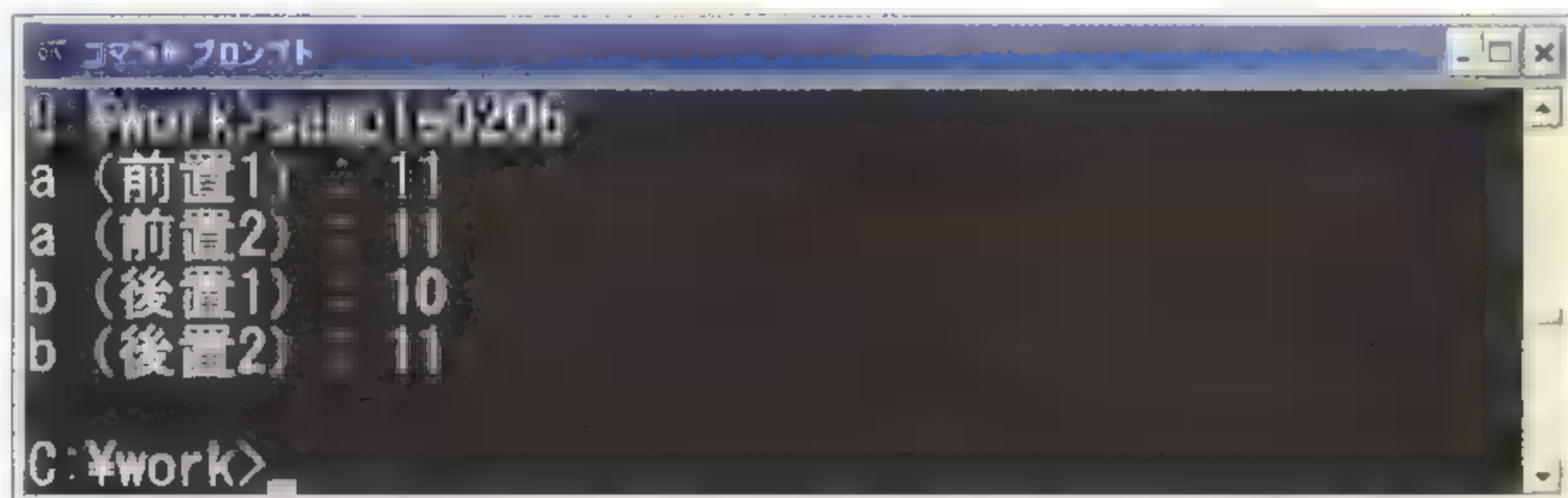
```
C:¥work>sample0205
a は 10 です。
b は 5 です。
a は 15 です。

C:¥work>
```

■ インクリメント演算子とデクリメント演算子

「**インクリメント演算子**」および「**デクリメント演算子**」とは、数値を1増やす（インクリメント）または1減らす（デクリメント）演算子のことです。インクリメント演算子およびデクリメント演算子は、前後のオペランドに作用し、前に置いた場合（前置）と後ろに置いた場合（後置）で挙動が異なります。

表3　インクリメント、デクリメント演算子

演算子	使用例	意　味
++	++a	（前置）変数aが使用される前に1が加算されます。
	a++	（後置）変数aが使用された後に1が加算されます。
−−	−−a	（前置）変数aが使用される前に1が減算されます。
	a−−	（後置）変数aが使用された後に1が減算されます。

演算子の「+」と「+」、「-」と「-」の間にスペースを入れてはいけません。また、前置と後置とでは、オペランドに値が加算・減算されるタイミングが異なります。次に前置と後置の違いを確かめるプログラムを示します。

Sample0206.c　インクリメントの利用

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int a = 10;
06      int b = 10;
07
08      printf("a (前置1) : %d¥n", ++a);
09      printf("a (前置2) : %d¥n", a);
10
11      printf("b (後置1) : %d¥n", b++);
12      printf("b (後置2) : %d¥n", b);
13
14      return 0;
15  }
```

前置は、「aに1加算」された後で、画面表示が実行されます。

後置は、画面表示が行われた後に、「bに1加算」されます。

5〜6行目では、初期値「10」を持つ**int**型の変数**a**と変数**b**を宣言しています。8行目では、前置のインクリメント演算子を利用しているため、変数**a**の値が出力される前に1が加算されます。逆に、11行目では、後置のインクリメント演算子を利用しているため、変数**b**の値が出力された後で1が加算されます。実行結果は次のとおりです。

なお、インクリメント演算子とデクリメント演算子は演算の対象となるオペランドが1つであるため、単項演算子と呼ばれます。

2. ビット演算子

「**ビット演算子**」とは、2進数表現のビット単位で値を操作できる演算子のことです。ビット演算子は、「ビット論理演算子」「ビットシフト演算子」の2つに大きく分けられます。

ただし、これらの演算子は入門レベルのプログラミングではほとんど利用しません(実際、本書でもこれ以降の項では登場しません)。予備知識とみなして「こういった演算子もあるのか」という認識にとどめておくだけでかまいません。

■ ビット論理演算子

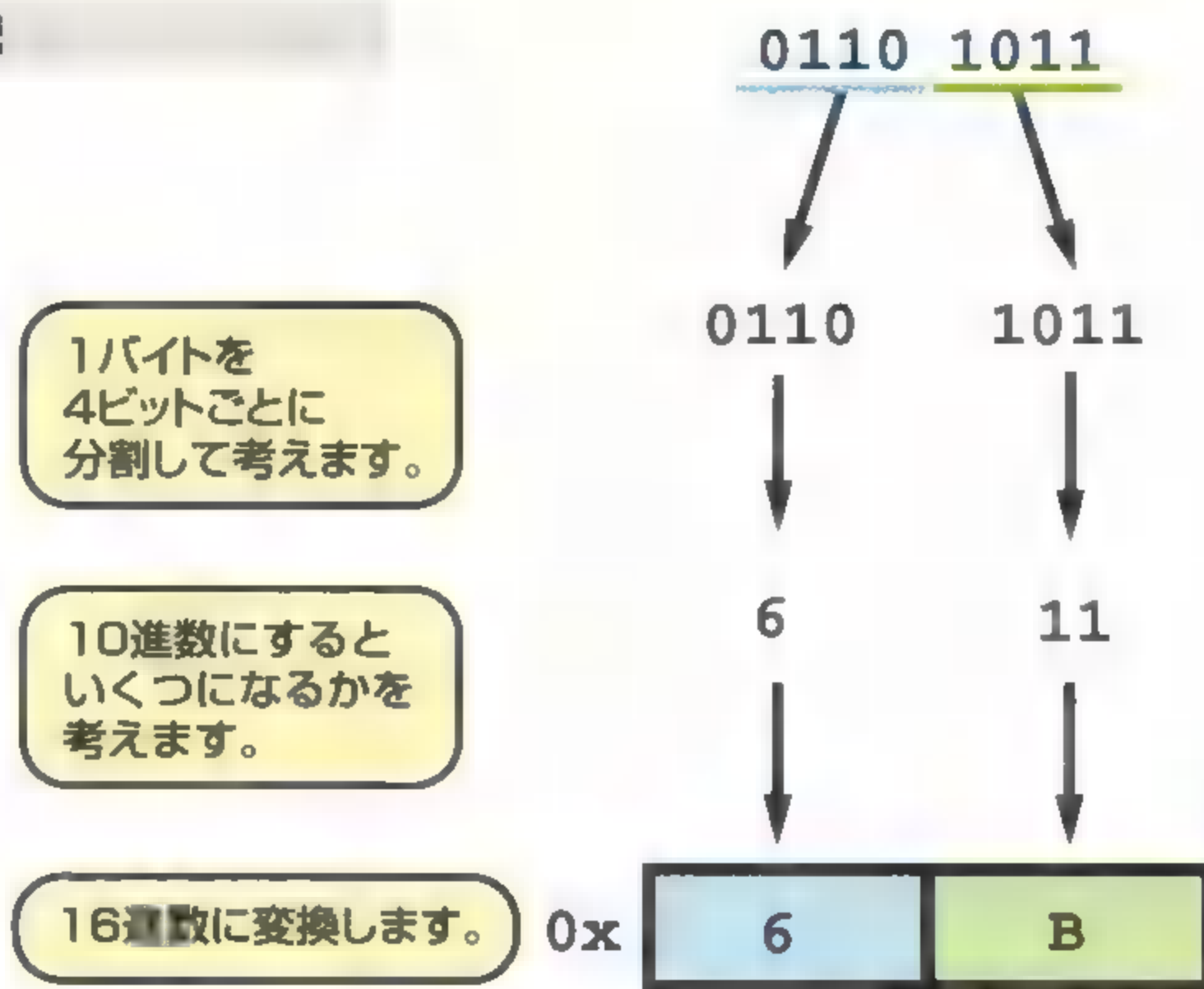
ビット論理演算子は、その名のとおり「ビットごとの0と1の演算」を行うための演算子です。演算子ごとの演算結果は、次のとおりです。

表4　ビット論理演算子

演算子	機　能	ビット演算結果
&	論理積(AND)	0 & 0 → 0
		1 & 0 → 0
		0 & 1 → 0
		1 & 1 → 1
|	論理和(OR)	0 | 0 → 0
		1 | 0 → 1
		0 | 1 → 1
		1 | 1 → 1
^	排他的論理和(XOR)	0 ^ 0 → 0
		1 ^ 0 → 1
		0 ^ 1 → 1
		1 ^ 1 → 0

ビットごとの演算は2進数表記で行うのがもっともわかりやすいのですが、C言語には数値を2進数で表現する方法がありません。そのため、一般的に16進数表記を利用してビット演算を記述します。これは、16進数表記の数値1つで4ビット分の情報が表現できるので、ビット長が長い値でも比較的短い文字数で表現できるという点と、16進数表記の数値が2つでちょうど1バイトになるという2つの点で便利であるためです。

2進数の数値を16進数で表記するには、次のように考えるとわかりやすいでしょう。

図2　16進数の表記

ビット論理演算子は、次のように使います。

Sample0207.c　ビットの論理演算

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int a = 0x6B;
06      int b = 0xA2;
07      int c = 0;
08      /* AND */
09      c = a & b;
10      printf("a & b = %x¥n", c);
11      /* OR */
12      c = a | b;
13      printf("a | b = %x¥n", c);
14      /* XOR */
15      c = a ^ b;
16      printf("a ^ b = %x¥n", c);
17      return 0;
18  }
```

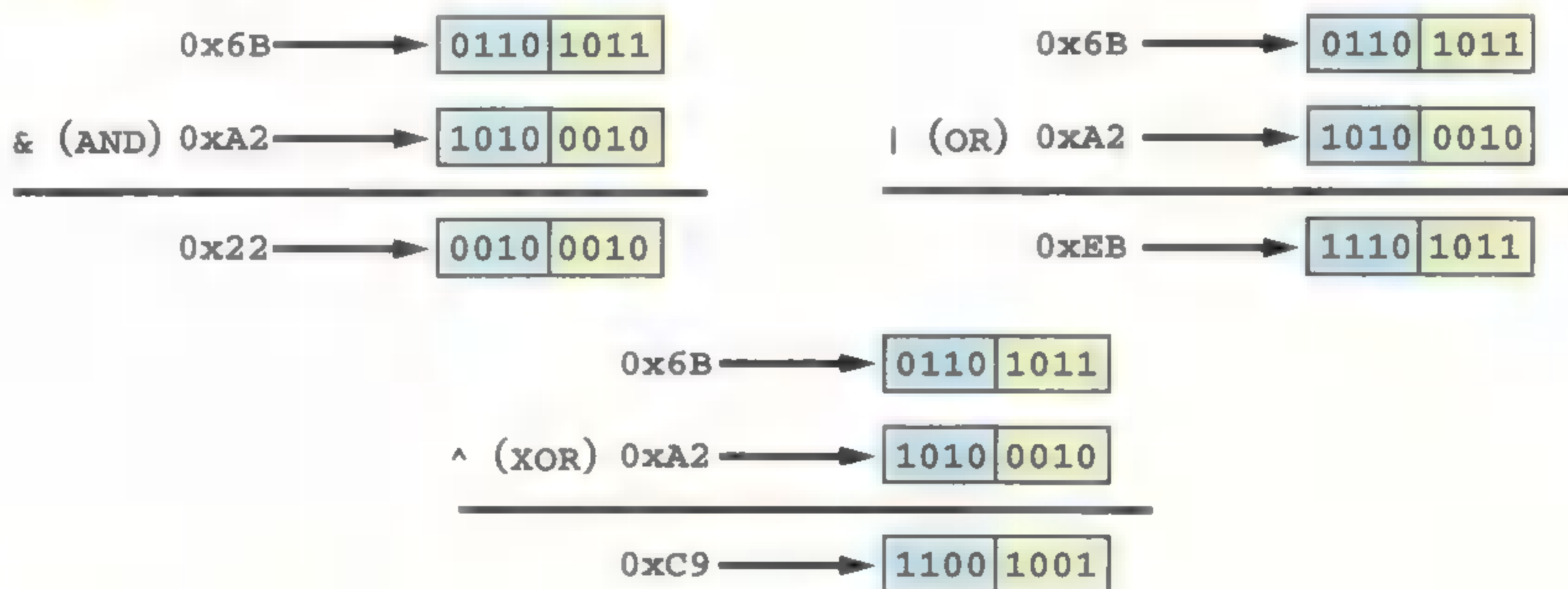
初期値を、16進数表記で記述しています。

AND演算の結果を代入しています。

OR演算の結果を代入しています。

XOR演算の結果を代入しています。

この例のビット演算は次のように行われます。実行結果から演算が正しく行われていることがわかります。

図3 ビット演算

■ ビットシフト演算子

ビットシフト演算子は、ビット情報を「**シフト**」する場合に利用します。「シフト」とは、変数のビット情報を右または左に「まるごと移動」することをいいます。たとえば、2進数で「0110」の値を右に1ビットシフトすると「0011」になります。

ビットシフト演算子には、次の2つがあります。

表5 ビットシフト演算子

演算子	機　能
<<	左にシフトします。
>>	右にシフトします。

左シフトを行うと、もっとも右のビットには自動的に「0」が補充されます。

右シフトを行う場合、シフトを行う変数の型によって挙動が変わります。符号なしの変数（`unsigned int`や`unsigned char`など）を右シフトする場合は、左シフトの場合と同じです。

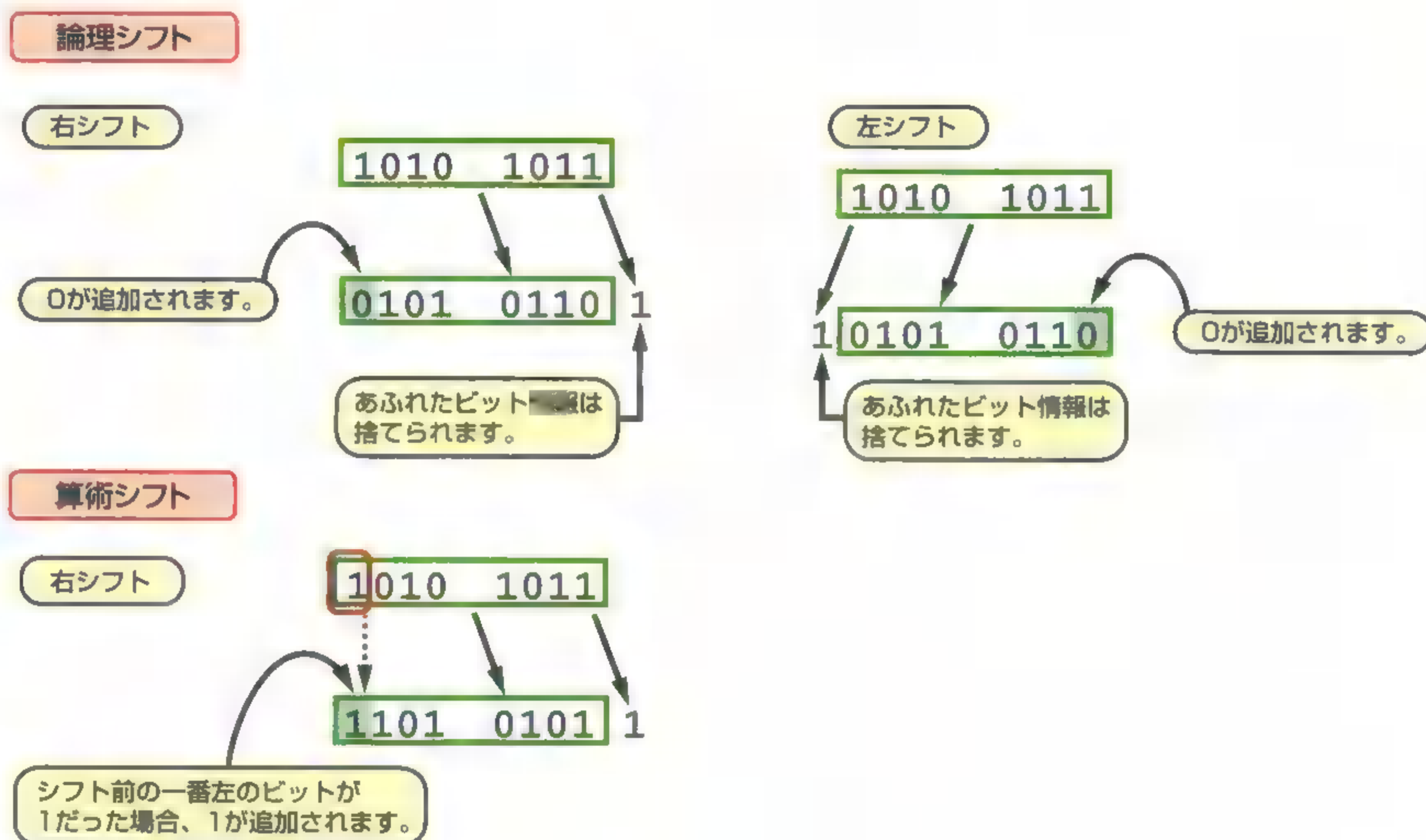
符号付きの変数型（intやcharなど）をシフトする場合、シフトする前の状態でもっとも左のビットが1のときに右シフトを行うと、補充されるビットには「1」が入ります。符号付き変数のもっとも左のビットは「符号」を表すビットとして利用されており、負の数値であれば、符号を維持するために、このような方法でシフトが行われます。これを「算術シフト」といいます。

これに対して、左シフトや符号なし変数の右シフトのことを「論理シフト」といいます。

図4 シフト演算

論理シフト

左シフト
1010 1011
1 0101 0110
0が追加されます。
あふれたビット情報は捨てられます。

算術シフト

Column

n進数のnは、何倍ごとに桁が上がるかを示す数字、つまり基数を表します。P.33の「8進数と16進数の代入」で説明したように、10進数では10倍ごと、8進数では8倍ごと、16進数では16倍ごとに桁が上がります。

同様に2進数では、2倍するたびに桁が上がります。2進数の0010を2倍すると0100、4倍すると1000になりますが、これは0010をそれぞれ左に2ビットおよび4ビットだけシフトした場合と同じ結果です。

逆に、2進数の1000を2で割ると0100、4で割ると0010になりますが、これは1000を右に2ビットおよび4ビットだけシフトした結果と同じになります。

つまり、ある値を左にnビットだけシフトすると2^n倍したことになり、右にnビットだけシフトすると2^nで割ったことになります。

ただし、この関係はシフトが行われてもその値の符号が維持される算術シフトにのみ適用されます。

Section 10

型の変換

覚えておきたいキーワード
- 型変換
- キャスト
- 演算子の優先順位

算術演算や代入演算の対象となるデータには、int型やdouble型などさまざまなものがあります。異なるデータ型どうしで演算や代入を行うと意図しない結果になることがあるため、データ型が異なる場合はデータ型を変換して統一します。

1. 型変換とキャスト

型変換とは...

C言語では、異なる型どうしの演算を指示された場合、自動的に「サイズの大きな型に合わせてから演算を行う」という処理を行います。

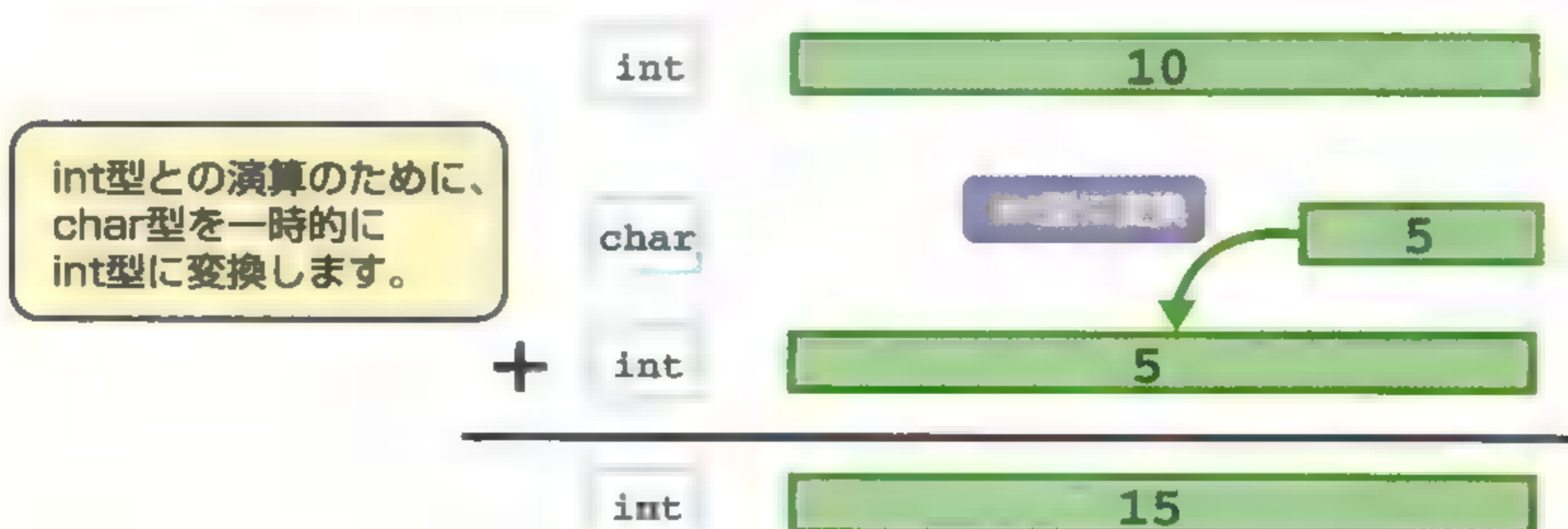
たとえば、次のような処理を記述した場合を考えてみましょう。

Sample0208.c　自動的な型変換

```
01  int a = 10;
02  char b = 5;
03  int c = a + b;
04  printf("c = %d", c);
```

int型(4バイト)とchar型(1バイト)で演算を行う場合、char型の変数を一時的に「int型」とみなして計算を行います。

実行結果からはわかりませんが、プログラムでは動作中に次に示すような処理が行われています。

図1　一時的な型変換

その結果として、次のように正しく演算が行われます。

■ キャストとは…

「**キャスト**」とは、型変換をプログラマが明示的に命令することをいいます。キャストを利用するのは、次のような場合です。

Sample0209.c　平均の演算（整数）

```
01  int a = 10;
02  int b = 5;
03  double c = (a + b) / 2; /* aとbの平均をcに代入する */
04  printf("c = %f", c);
```


この例では変数**a**と変数**b**の平均を求めて変数**c**に代入したつもりですが、正しい結果が得られません。これは変数**a**と変数**b**が**int**型で、数値「2」も整数型とコンパイラに判断されるので、式の演算結果も**int**型になります。**int**型は小数点を含む数値を扱えないため、結果が「7」になってしまいました。

このような場合、正しく計算を行うためにキャストを行います。キャストの構文は次のとおりです。

構文　キャスト

```
(型名) 変数名;
```

キャストを利用して上のサンプルを修正すると、次のようになります。

Sample0210.c　平均の演算（キャスト）

```
01  int a = 10;
02  int b = 5;
03  double c = (double) (a + b) / 2.0;  /* aとbの平均をcに代入する */
04  printf("c = %f", c);
```

int型のデータをdouble型にキャストして計算することで、

変数aとbを足した結果をキャストして一時的に**double**型として扱うように命令しているため、コンパイラは正しく浮動小数点数型の演算を行うことができます。

Column 演算子の優先順位

式の中に複数の演算子がある場合には、下表のように定められた演算子の優先順位にもとづいて、優先順位の高いものから演算が行われていきます。優先順位が同じ場合には、演算子ごとに式の左右のどちらから演算を行うかが決められています。決められた優先順位と違う順番で演算を行いたい場合には、先に演算を行いたい部分を( )でくくります。( )は多重に指定することもできます。

表1　演算子の優先順位

優先順位	文法関係	算術演算子・論理演算子など	ビット演算子	同レベルの場合の計算方向
1	() [] -> . ++ --			→ 左から右へ
2		! + −(正負の符号)	~	← 右から左へ
3	キャスト			← 右から左へ
4		* / %		→ 左から右へ
5		+ −		→ 左から右へ
6			<< >>	→ 左から右へ
7		> >= < <=		→ 左から右へ
8		== !=		→ 左から右へ
9			&	→ 左から右へ
10			^	→ 左から右へ
11					→ 左から右へ
12		&&		→ 左から右へ
13		||		→ 左から右へ
14		? :		← 右から左へ
15		= += -= *= /= %=	&= |= ^= <<= >>=	← 右から左へ

Column 浮動小数点数の剰余を求める

「%」は剰余を求める演算子であると説明しましたが、浮動小数点数に対してこの演算子を利用することはできません。この演算子は「整数どうし」の除算による剰余を求めます。

```
int n = 5 % 3;
double d = 5.0 % 3.0
```

整数どうしは、剰余を求めることができます。

浮動小数点数は、剰余を求めることができません（コンパイルエラー）。

浮動小数点数の剰余を求めるには、キャストを利用します。剰余の演算では、演算に用いる値が2つとも整数である必要があるので、両方の浮動小数点数についてキャストを行います。

浮動小数点数の剰余の演算を行うプログラムは、次のようになります。

Sample0211.c　浮動小数点数の剰余

```
01  #include <stdio.h>
02
03  int main()
04  {
05      double a = 11.0;
06      double b = 3.0;
07      int c = (int)a % (int)b;    /* 剰余の演算 */
08      printf("c = %d", c);
09
10      return 0;
11  }
```

まとめ　第2章：変数と演算子

この章では、変数の宣言や演算の方法、型の変換などデータを扱うための基礎を学びました。また、ビットとバイトなど、プログラミングを行う上で重要な知識も学びました。データ型と、その型で扱える数値の幅や必要バイト数は今後も必要となる知識であるため、しっかりと覚えておきましょう。

第2章で学習したこと

- プログラムで利用する値を記憶するためには、変数を利用する。
- 「0」か「1」の組み合わせで大きな数値を表現する。
- 1つの0または1のことを「ビット」といい、ビットが8個集まったものを「バイト」という。
- それぞれの文字には「文字コード」という数値が割り当てられている。たとえば、文字「A」には値「65」が割り当てられている。
- printf()関数で変数を表示するためには、変換指定子を利用する。また、改行などを行うためにはエスケープシーケンスを利用する。
- 演算子を使うと、変数に値を代入したり、加減乗除を行うなどさまざまな処理（演算）を行うことができる。
- データ型が異なる変数の演算では、意図しない結果になることがある。それを防ぐためには、型変換（キャスト）を行う。

ステップアップ！

変数や演算子、そしてビットやバイトなどの知識は、プログラミングを行う上でもっとも重要な知識といってよいでしょう。特に変数の扱える値の範囲や、演算の順番などをしっかり把握することは重要です。これらを勘違いしてコーディングを行うと、原因のわかりにくいバグとなる可能性があるからです。解説を読んでいくうちに疑問が生じたときには、自分でプログラムを作成して実際に試してみましょう。

問1　演算と表示

変数に「500 + 150」の演算結果を代入し、その変数を画面に表示してください。

答1

変数に値を代入するには「**=**」を利用します。また整数を表示するための変換指定子は「**%d**」を利用します。

```
#include <stdio.h>

int main()
{
    int n = 500 + 150;
    printf("%d", n);
    return 0;
}
```

問2　文字コード

char型の変数に文字「Z」の文字コードを代入し、その変数を文字として画面に表示してください。また、その変数に代入されている文字コードを数値として、あわせて画面に表示してください。

答2

文字コードは実際はただの数値です。そのため、**printf()**関数で表示する際に文字として扱う変換指定子（**%c**）を記述すると文字として表示され、数値として扱う変換指定子（**%d**）を記述すると数値として表示されます。

```
01  #include <stdio.h>
02
03  int main()
04  {
05      char c = 'Z';
06      printf("文字: %c ¥n", c);
07      printf("数値: %d ¥n", c);
08      return 0;
09  }
```

変数cを「文字」として表示します。

変数cを「数値」として表示します。

問3 キャスト

2つの整数の値「10」と「7」の平均を求め、**double**型の変数に代入し、小数点以下まで正しく表示してください。

答3

整数の値どうしの演算では小数点以下まで処理が行われないため、**double**型にキャストして演算を行います。

```
int a = 10;
int b = 7;
double c = (double) (a + b) / 2.0;
```

第3章

Visual Learning Introduction of C

制御構文

条件判断

覚えておきたいキーワード
- 条件
- if～else文
- switch文

プログラムは原則的に上から順番に実行されますが、制御構文を利用すると、この流れを分岐させたり、繰り返させたりして制御することができます。制御構文は、「正しい」か「正しくない」かの2通りの条件をもとにプログラムの流れを制御します。

1. 制御構文と条件

■ 制御構文とは...

たとえば、「数学の点数が80点以上であれば合格、さもなければ不合格と画面に表示する」プログラムを作成するには、「**条件によって異なる処理**」を行う必要があります。このように処理の流れを制御するには「**制御構文**」を利用します。

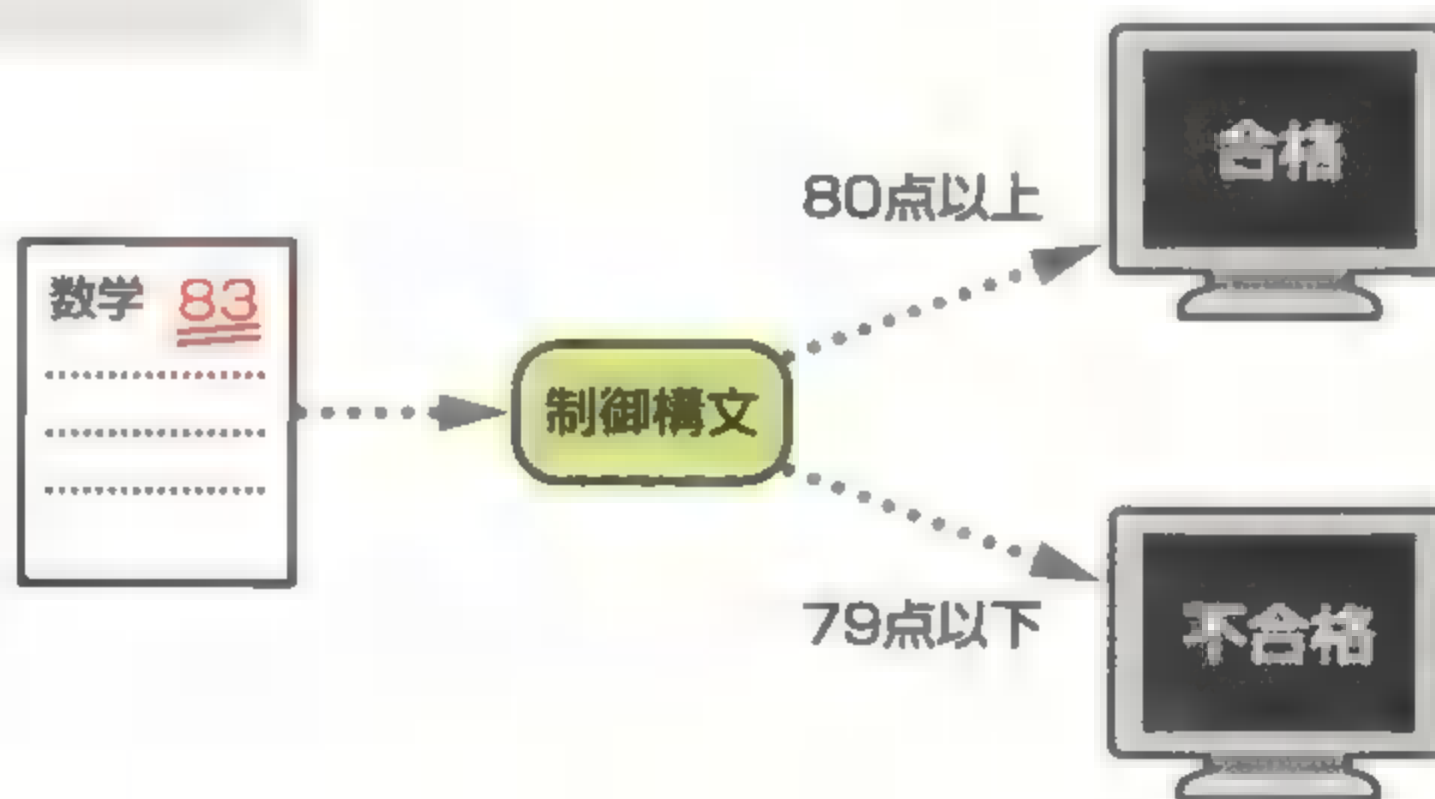
図1 制御構文

C言語の制御構文には次のようなものがあります。

表1 制御構文の種類

種類	内容
if	条件が正しい場合のみ処理を行います。
if～else	条件が正しい場合と正しくない場合で異なる処理を行います。
switch	値によって複数の処理を振り分けます。
for	条件を満たす間、処理を繰り返します。
while	条件を満たす間、処理を繰り返します。
do～while	条件を満たす間、処理を繰り返します(最低1回は処理を行います)。

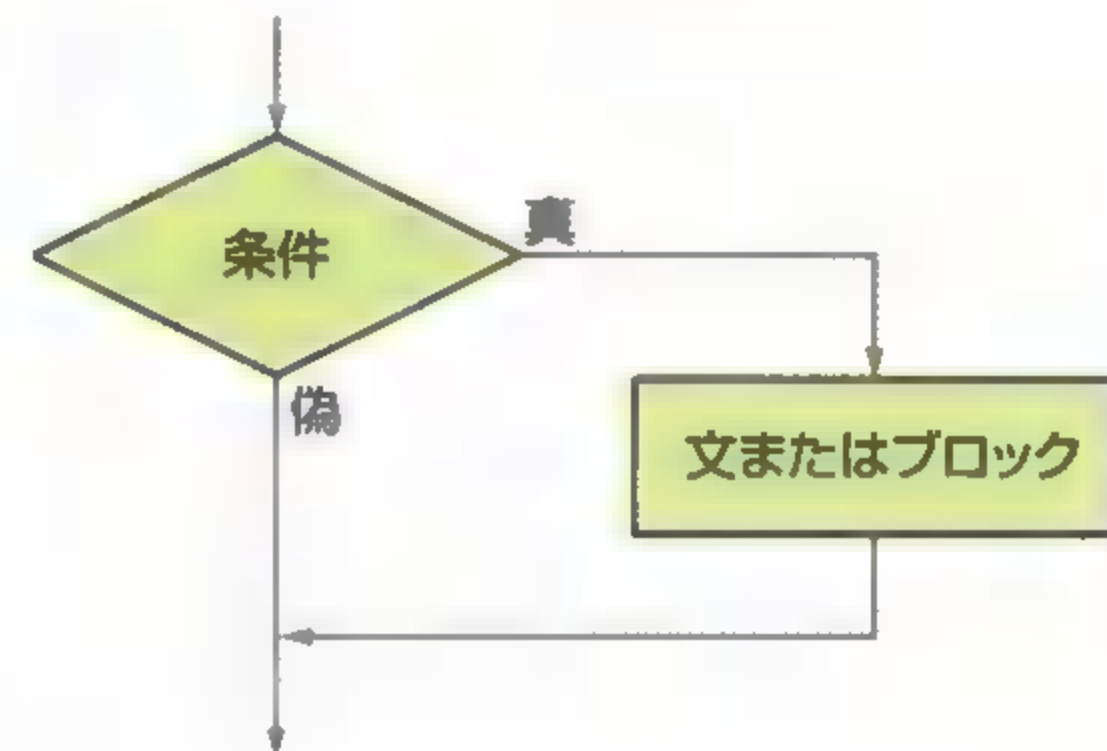
■ 条件とは…

先ほどの例でいうと、「数学の点数が80点以上であれば」という部分が「**条件**」にあたります。また、条件を満たしているかどうかを判断することを「**評価**」といいます。評価の結果は「真（条件を満たしている）」か「偽（条件を満たしていない）」で表現されます。

C言語では真偽を数値で表現し、「**真**」は「**0以外の数値**」、「**偽**」は「**0**」を意味します。

2. if文

■ if文とは…

「**if**文」とは、指定した式の評価の結果が真か偽かで処理を分岐させるための制御構文です。**if**文の書式は次のとおりです。

条件式に記述した内容が真（つまりカッコの中の演算結果が「0以外の数値」）である場合、**if**文に続けて記述した処理が実行されます。偽（演算結果が「0」）である場合、何も処理は実行されません。

ブロックを用いない場合、条件が真であるときに行われる処理は**if**文の直後に書かれた1文だけです。ブロックを用いる場合は、**if**文に続くブロックの中の処理がすべて行われます。

なお、**if**文で処理を分岐させた後の処理が1文だけである場合でも、ソースコードをわかりやすくするためにブロックを利用するのが一般的です。

■ 条件を判断する演算子

if文の()の中に条件式を記述するための演算子には次の2種類があります。

(1) 関係演算子

2つの数値の大小関係を調べる演算子のことです。記述した式の内容が正しい場合は真、間違っている場合は偽が演算結果として返されます。

表2　関係演算子

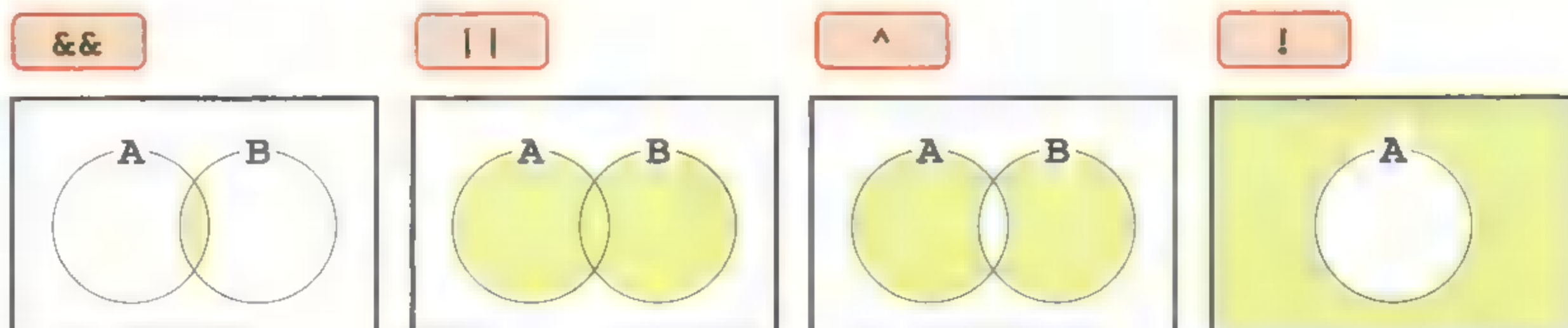
関係演算子	使用例	意　味
==	a == 7	a と7が等しいか?
!=	a != 7	a と7が等しくないか?
>	a > 7	a が7よりも大きいか?
>=	a >= 7	a が7以上か?
<	a < 7	a が7よりも小さいか?
<=	a <= 7	a が7以下か?

(2) 論理演算子

複数の条件を組み合わせたい場合に利用する演算子のことです。たとえば、「変数aが『3以上』で、かつ『7以下』か?」という判断を行う場合に利用します。

表3　論理演算子

演算子	使用例	意　味
&&	A && B	論理積(～かつ～)。A と B の両方が真の場合、真になります。
||	A || B	論理和(～または～)。A と B の少なくとも一方が真の場合、真になります。
^	A ^ B	排他的論理和。A と B の一方だけが真の場合、真になります。
!	!A	否定。A の真と偽を逆にします。

図3　各演算子の条件

これらの演算子を利用して、**if**文の**()**の中に条件式を記述します。

たとえば、冒頭の「数学の点数が80点以上であれば」という条件を**if**文で表現すると、次のようになります。ここでは、数学の点数を変数**math**で表現します。

```
if( math >= 80 )
```

また、「数学の点数が80点以上、90点以下であれば」という条件を**if**文で表現すると、次のようになります。

「>=」「<=」演算子による比較を「&&」演算子より先に行うため、( )で囲みます。

```
if( ( math >= 80 ) && ( math <= 90 ) )
```

次に示すのは、**if**文を利用して「変数**math**が80以上である場合に特定のメッセージを表示する」プログラムです。

Sample0301.c if文による条件判断

```
01 #include <stdio.h>
02
03 int main()
04 {
05     int math = 70;
06     printf("math : %d¥n", math);
07     if(math >= 80){
08         printf("%d点は合格です!¥n", math);
09     }
10
11     math = 90;
12     printf("math : %d¥n", math);
13     if(math >= 80){
14         printf("%d点は合格です!¥n", math);
15     }
16     return 0;
17 }
```

「変数mathが80以上であれば」という条件を表します。

7行目のif文では変数mathの値が「70」であるため、条件の「math >= 80」を満たしません。そのため、if文のブロック内の処理が行われていないことが実行結果から確認できます。

13行目のif文では変数mathの値が「90」であるため、条件の「math >= 80」を満たします。そのため、if文のブロック内の処理が実行されています。

3. if～else文

■ else文とは...

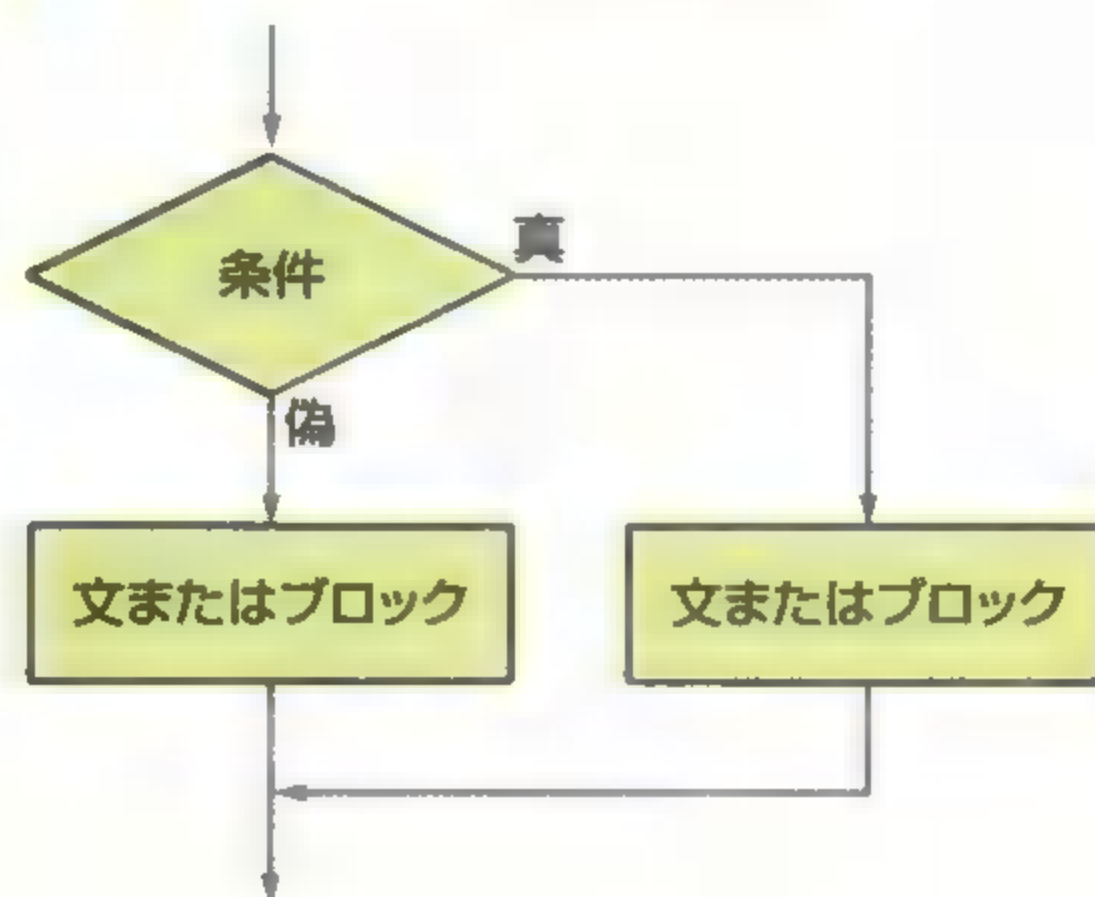
if文では、指定した条件を満たす（真である）場合の処理だけを記述できました。else文は、条件を満たさない（偽である）場合の処理を記述するために利用します。if～else文の書式は次のとおりです。

構文 if～else文

```
if( 条件式 )
    ……条件式が真のときの処理……;
else
    ……条件式が偽のときの処理……;
```

構文 if～else文（ブロックの利用）

```
if( 条件式 ) {
    ……条件式が真のときの処理……;
}
else {
    ……条件式が偽のときの処理……;
}
```

図4 if～else文

if文の直後に「条件を満たす（真である）場合の処理」を記述するのは同様ですが、その後に続けて**else**文と、「条件を満たさない（偽である）場合の処理」を記述します。

else文を利用したプログラムは、次のようになります。

sample0302.c　**if～else文の利用**

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int math = 70;
06      printf("math : %d¥n", math);
07      if(math >= 80){
08          printf("%d点は合格です!¥n", math);
09      }
10      else {
11          printf("%d点は不合格です。¥n", math);
12      }
13
14      math = 90;
15      printf("math : %d¥n", math);
16      if(math >= 80){
17          printf("%d点は合格です!¥n", math);
18      }
19      else {
20          printf("%d点は不合格です。¥n", math);
21      }
22      return 0;
23  }
```



```
C:¥work>sample0302
math : 70
70点は不合格です。
math : 90
90点は合格です!

C:¥work>
```

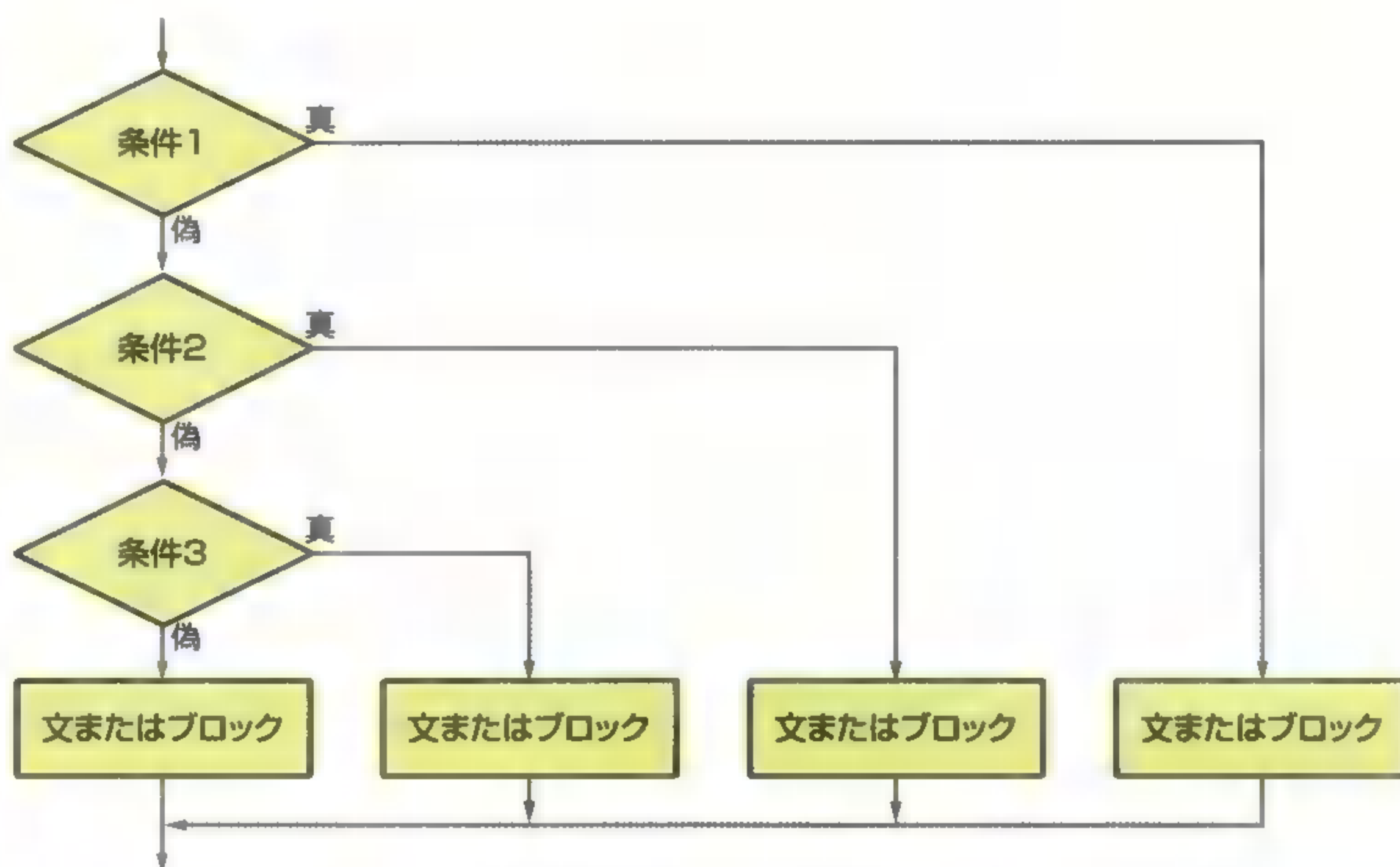
7行目の**if**文では変数**math**の値が「70」であるため、条件の「**math >= 80**」を満たしません。そのため、**else**ブロック内の処理が実行され、「不合格です。」と画面に表示されます。

16行目の**if**文では変数**math**の値が「90」であるため、条件の「**math >= 80**」を満たします。そのため、**if**ブロック内の処理が実行され、「合格です!」と画面に表示されています。

■ if～else文を続けて記述する

判断したい条件がいくつかある場合、**if～else**文を続けて記述することで複数の条件を続けて判断することができます。

図5 連続したif～else文

条件1 —真→ 文またはブロック
偽↓
条件2 —真→ 文またはブロック
偽↓
条件3 —真→ 文またはブロック
偽↓
文またはブロック

このように処理を振り分けるには、次のように**if～else**文を続けて記述します。

構文 連続したif～else文

```
if( 条件1 )
    ……条件1が真のときの処理……;
else if( 条件2 )
    ……条件2が真のときの処理……;
else if( 条件3 )
    ……条件3が真のときの処理……;
    ⋮
else
    ……すべての条件が偽のときの処理……;
```

連続したif〜else文（ブロックの利用）

```
if( 条件1 ) {
    ……条件1が真のときの処理……;
}
else if( 条件2 ) {
    ……条件2が真のときの処理……;
}
else if( 条件3 ) {
    ……条件3が真のときの処理……;
}
    ⋮
else {
    ……すべての条件が偽のときの処理……;
}
```

この場合、一番上の**if**文から順に条件が評価され、結果が真となった**if**文に続くブロックの処理が実行されると、処理は一連の**if**〜**else**文を抜けます。

たとえば、条件1の条件判断の結果が偽であれば**else**文に処理が分岐し、次の「**if**(条件2)」の評価が実行されます。条件2を評価して結果が真であれば「条件2が真のときの処理」を実行し、その後「**if**(条件3)」には移行しません。

次に示すのは、**if**〜**else**文を続けて記述し、複数の条件を判断するプログラムです。

Sample0303.c if〜else文を連続させる例

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int math = 85;
06      printf("math : %d¥n", math);
07      if(math >= 90){
08          printf("%d点は優秀です!¥n", math);
09      }
10      else if(math >= 80){
11          printf("%d点は良くできています。¥n", math);
12      }
13      else if(math >= 70){
```

1 変数mathは85なので、
2 この条件を満たしません。
3 else文に処理が移り、
4 else文に続く処理として、このif文が実行されます。
5 条件を満たすので、この処理が実行されます。

```
14            printf("%d点はまぁまぁです。¥n", math);
15        }
16        else {
17            printf("%d点はもっと頑張りましょう。¥n", math);
18        }
19        return 0;
20    }
```

```
C:¥work>sample0303
math : 85
85点は良くできています。

C:¥work>_
```

変数**math**には85が代入されているため、7行目の**if**文の条件は満たしません。従って、10行目の**else**に処理が分岐します。**else**文に続いて**if**文があるため、条件が評価されますが、今度は条件を満たすので11行目の処理が行われます。その後の**else**以降の処理は行われません。

4. switch文

■ switch文とは…

if文と同様に、条件によって処理を振り分ける制御構文として「**switch**文」があります。**if**文は条件の評価結果が真か偽かに応じて2つの処理に振り分けますが、**switch**文は条件の値に応じて複数の処理に振り分けることができます。

switch文の書式は右頁のとおりです。

switch文では、「条件式」の演算結果と**case**の「値」を比較し、等しければそれ以降に記述された処理を実行します。処理は「**break**」が記述されている行まで実行されます。前述の例でいえば、条件式の結果が「値1」である場合は「処理1」が実行されるといった具合です。

if文と違い、**case**の後に記述した文はブロックにしなくても**break**が記述された行に到達するまで順番に実行されます。

caseの値のうち、条件式の演算結果と等しいものがない場合、「**default**」に続いて記述した処理が実行されます。**default** は省略することもできます。

構文 switch文

```
switch ( 条件式 ) {
    case 値1:
        ……処理1……;
        break;
    case 値2:
        ……処理2……;
        break;
    default:
        ……処理3……;
        break;
}
```

次に示すのは、**switch**文を利用したプログラムです。

switch文の条件式である変数**val**には、文字「B」が保存されています。そのため、ここでは10行目の「**case 'B':**」から12行目の「**break;**」までの処理だけが実行されます。

Sample0304.c switch文の利用

```
01 #include <stdio.h>
02
03 int main()
04 {
05     char val = 'B';
06     switch( val ){
07         case 'A':
08             printf("valはAです。");
09             break;
10         case 'B':
11             printf("valはBです。");
12             break;
13         default:
14             printf("valはAでもBでもありません。");
15             break;
16     }
17     return 0;
18 }
```

valと等しい値のcase文があるかを評価します。

valが'A'と等しい場合の処理を記述します。

1 この場合valは'B'と等しいので、

2 この処理が行われます。

どのcase文にも等しい値がない場合に実行されます。

■ switch文の注意点

switch文には、次のような注意点があります。

(1) 条件式の結果は、整数でなければならない

switch文の条件には評価結果が整数になる式を指定する必要があります。

(2) caseに続く値には、変数や式を利用できない

条件式との比較の対象となる**case**の値には、変数や式を指定することはできません。

誤った例

```
int good = 90;
int math = 75;
switch( math ) {
    case good:
    ⋮
```

変数は指定できません。

```
int good = 90;
int math = 75;
switch( math ) {
    case (math < good):
    ⋮
```

式は指定できません。

正しい例

```
int math = 75;
switch( math ) {
    case 90:
      ⋮
```

数値を直接指定しています。

(3) caseと値の後には「;(セミコロン)」ではなく「:(コロン)」を記述する

caseには条件式と比較する値を指定した後、コロンに続いて条件が一致した場合に実行する処理を記述します。

```
case 90;
```

正しい例

```
case 90:
```

(4) break文の記述を忘れると処理が終わらない

caseの値に応じて異なる処理を実行させたい場合に、処理の最後に**break**文を記述し忘れると、**break**文が現れるか**switch**文の終わりに到達するまでその後に続く処理がすべて実行されてしまいます。

```
int math = 90;
switch( math ){
    case 90:
        printf("90点です。¥n");
    case 85:
        printf("85点です。¥n");
        ⋮
```

1 break文を記述していないので、

2 上の処理から続けて、この処理まで実行されてしまいます。

変数**math**には90が代入されているので、「**case 90:**」の条件にあてはまります。よって画面には「90点です。」と表示されるのですが、**case**文の終わりを表す**break**文がありません。この場合、コンパイルエラーにはならず、「この後の**case**文も続けて実行する」という意味になります。

このプログラムを実行すると、下のように画面に「90点です。」と「85点です。」が両方とも表示されてしまいます。

ただし、次のプログラムのように、わざと**break**文を記述しない場合もあります。

```
int math = 90;
switch( math ){
    case 100:
    case 90:
        printf("あなたは大変優秀です。¥n");
        break;
    case 80:
    case 70:
        printf("試験は合格です。¥n");
        break;
}
```

1 100点と90点の場合は。
2 この処理を行います。
3 80点と70点の場合は。
4 この処理を行います。

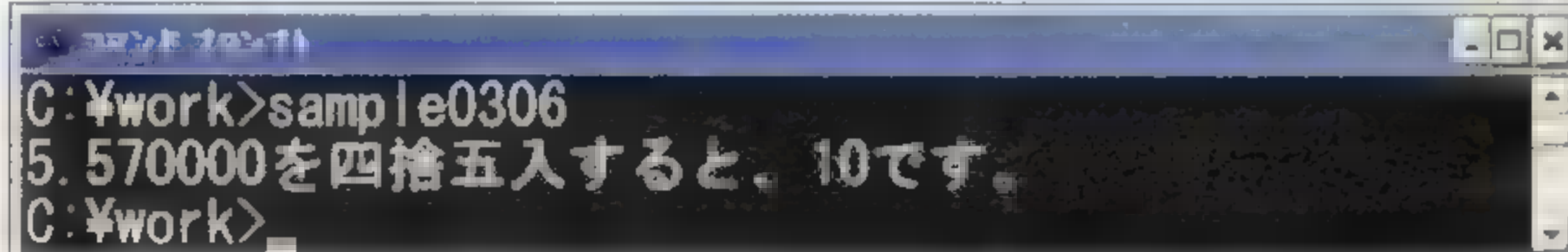
このように**break**文を記述しないことで、複数の条件のいずれかひとつにあてはまる場合の処理を記述することができます。

Column 三項演算子

ここまで、if文やswitch文を利用して条件によって異なる処理を行ってきました。しかし、条件や処理が簡単な場合には、次の条件演算子を利用することもできます。

構文 三項演算子

```
条件 ? 条件が真の場合の処理 : 条件が偽の場合の処理
```

条件演算子は「?」の前の条件を評価し、真の場合は「:(コロン)」の前の処理を行い、偽の場合は「:(コロン)」の後ろの処理を行って、演算結果として返します。

次に条件演算子を利用したプログラムを示します。

Sample0306.c 三項演算子

```
01  #include <stdio.h>
02
03  int main()
04  {
05      double d = 5.57;
06      int val;
07
08      val = (d >= 5.0) ? 10 : 0;
09      printf("%fを四捨五入すると、%dです。", d, val);
10
11      return 0;
12  }
```

dが5.0以上であれば10、それ以外の場合は0を返します。

```
C:¥work>sample0306
5.570000を四捨五入すると、10です。
C:¥work>_
```

このプログラムでは条件演算子を利用して、変数dが5.0より大きければ10を、5.0未満のときは0を、int型の変数valに代入しています。変数dに5.57が代入されているため、変数valには10が代入されます。なお、この条件演算子と同様の処理をif~else文で行うと右のようになります。

```
if(d >= 5.0) {
    val = 10;
}
else {
    val = 0;
}
```

覚えておきたいキーワード

- for文
- while文
- do～while文

繰り返し処理

繰り返し処理のことを「ループ」といいます。ループを行う制御構文を利用すると、ある条件を満たす間、同じ処理を繰り返すことができます。ループを行う制御構文には、for文やwhile文、do～while文などがあり、目的に応じて適切なものを選択する必要があります。

1. for文の利用

for文とは…

「**for**文」は、特定の条件を満たす間、処理を繰り返す制御構文です。一般に、カウンタと呼ばれる変数を使って、処理の内容や対象を切り替えながら特定の回数だけ処理を繰り返す目的でよく使われます。**for**文の書式は次のとおりです。

構文 for文

```
for (初期化の式; 継続条件; 増減の式) 命令文;
```

構文 for文（ブロックの利用）

```
for (初期化の式; 継続条件; 増減の式) {
    …… 繰り返したい処理 ……
}
```

(1) 初期化の式

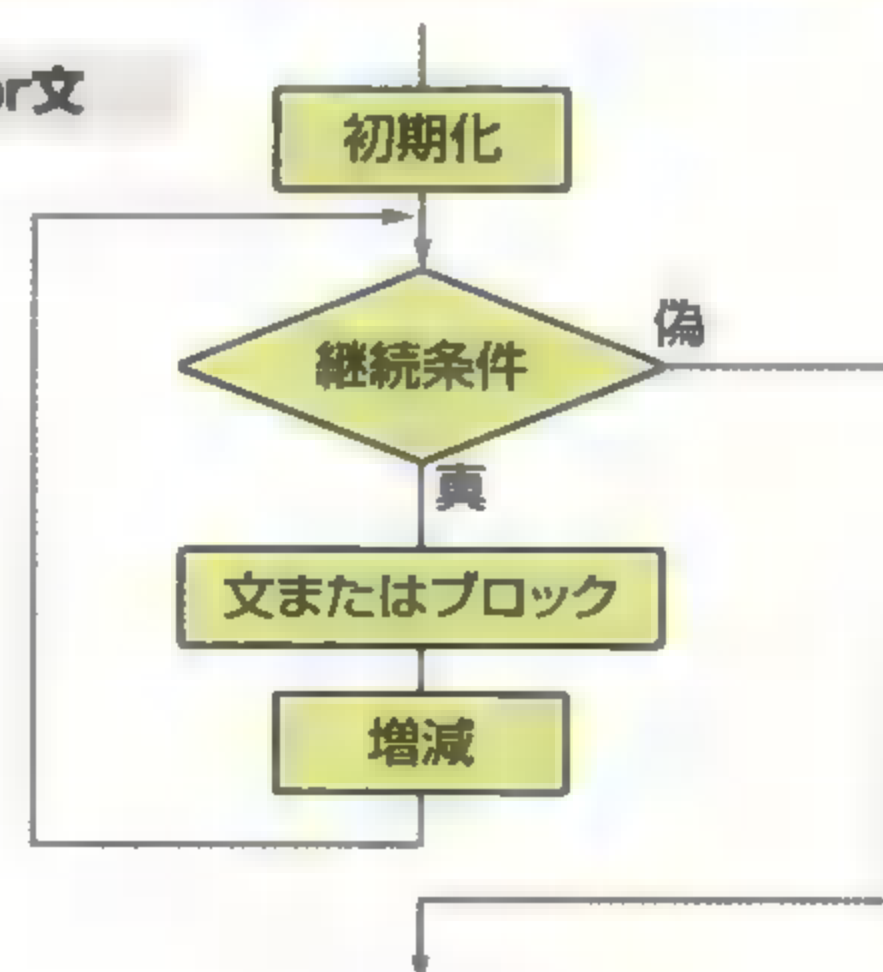
カウンタの初期値を指定します。変数の宣言も同時に行うことができます。

(2) 継続条件

処理を繰り返すための条件を指定します。

(3) 増減の式

カウンタの増減の方法を指定します。

図1 for文

forで文では、まず「初期化の式」を実行します。次に、「継続条件」を評価し、真であれば命令文またはブロック内の処理を実行した後、「増減の式」を実行します。以降「継続条件」が偽になるまで同様の処理を繰り返し、偽になったらfor文の処理を終了します。

カウンタとなる変数の名前は、一般的にiやj、kなど短いものが利用されます。また、「初期化の式」「継続条件」「増減の式」はそれぞれ省略することも可能です。

次に示すのは、整数を0から順に足して画面に表示していくプログラムです。

Sample0307.c　for文の利用

```
01 #include <stdio.h>
02
03 int main()
04 {
05     int i;
06     int sum = 0;
07     for (i=0; i<10; i++) {
08         sum += i;
09         printf("i:%d, 合計:%d¥n", i, sum);
10     }
11     return 0;
12 }
```

カウンタiの初期値を0に設定しています。

カウンタiが10未満の間処理を繰り返します。

ブロック内の処理を行うたびにカウンタiをインクリメントします。

このプログラムでは、for文のカウンタに整数型の変数iを利用し、初期値として0を代入しています。for文内の処理を行うたびに、カウンタiに1ずつ加算し（「i++」）、これをカウンタiが10未満の間繰り返します（「i < 10」）。結果として、for文内の処理が10回繰り返されます。このプログラムを実行した結果を次に示します。

2. while文

■ while文とは…

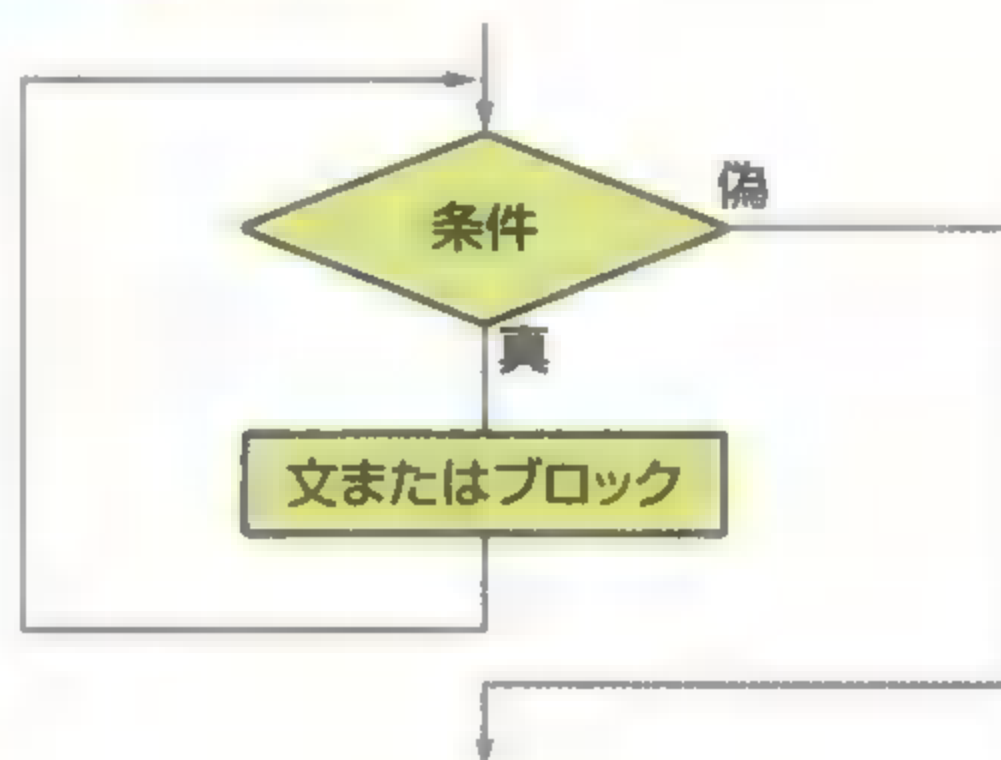
「**while**文」は特定の条件を満たす間、処理を繰り返す制御構文です。たとえば、「合計が50以下の間は足し算を繰り返す」というように、繰り返す回数はわからないが、繰り返す条件は決まっている場合に利用します。

while文の書式は次のとおりです。

構文 while文

```
while( 条件式 ) 命令文;
```

構文 while文（ブロックの利用）

```
while( 条件式 ) {
    …… 繰り返したい処理 ……
}
```

while文を利用すると、条件が偽になるまで命令文またはブロック内の処理を繰り返します。また、初めから条件が偽となる場合は、一度も処理を実行せずに**while**文の次に処理が移行します。

また、「条件」に常に真となる式（たとえば「1」など）を指定すると、ループがいつまでも終了しない無限ループとなるため注意が必要です（P.80参照）。

次に示すのは、「整数を1から順に合計が50より大きくなるまで足して画面に表示していく」プログラムです。

Sample0308.c　while文の利用

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int sum = 0;
06      int i = 1;
07
08      while(sum <= 50) {
09          sum += i;
```

合計を表す変数sumが50以下の間、処理を繰り返します。

```
10          printf("i:%d, 合計:%d¥n", i, sum);
11          i++;
12      }
13      return 0;
14  }
```

このプログラムでは、**while**文の条件に「**sum <= 50**」を指定し、合計を格納した変数**sum**が50以下の場合は、**while**文内の処理を繰り返すようにしています。10回処理を繰り返すと、変数**sum**の値が「55」となり条件を満たさなくなるため、**while**文の処理が終了します。

このプログラムを実行した結果を次に示します。

```
C:¥work>sample0308
i:1, 合計:1
i:2, 合計:3
i:3, 合計:6
i:4, 合計:10
i:5, 合計:15
i:6, 合計:21
i:7, 合計:28
i:8, 合計:36
i:9, 合計:45
i:10, 合計:55

C:¥work>
```

3. do～while文

■ do～while文とは…

「**do～while**文」は**while**文と同様に、特定の条件を満たす間、指定した処理を繰り返し実行する制御構文です。**while**文と異なる点は、処理を行ってから継続条件を評価するため、最低でも1回は**do～while**文のブロック内(もしくは命令文)の処理を行う点です。

do～while文の書式は次のとおりです。

構文 do～while文

```
do 命令文; while( 条件式 );
```

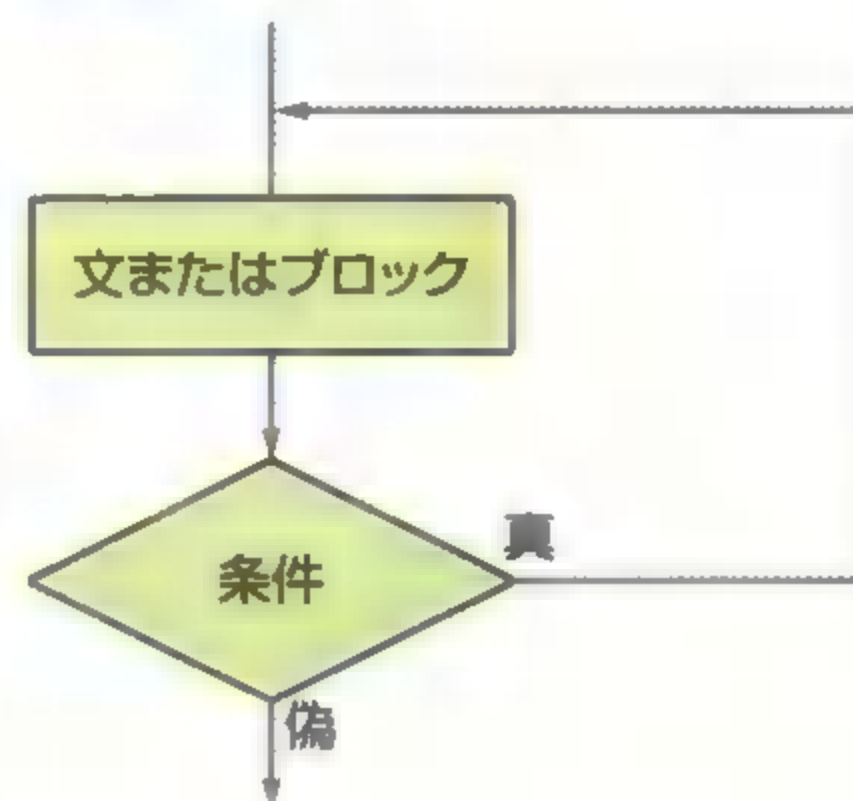
構文 do〜while文（ブロックの利用）

```
do {
    …… 繰り返したい処理 ……
} while( 条件式 );
```

do〜while文を利用すると、条件が偽になるまで命令文またはブロック内の処理を繰り返します。特に、**do〜while**文では「**;**（セミコロン）」の位置に注意してください。なお、ブロックを利用しない**do〜while**文はわかりにくいため、あまり利用されません。

また、「条件」に常に真となる値を指定すると、ループがいつまでも終了しない無限ループとなるため注意が必要です（P.80参照）。

図3 do〜while文

do〜while文の特徴を示したプログラムは、次のようになります。

Sample0309.c do〜while文の特徴

```
01 #include <stdio.h>
02 
03 int main()
04 {
05     int sum = 0;
06     int i = 1;
07 
08     do {
09         sum += i;
10         printf("i:%d, 合計:%d¥n", i, sum);
11         i++;
12     } while (sum < 0);
13     return 0;
14 }
```

変数sumの初期値は「0」であるため、初めから条件を満たしませんが、ループの1回目の処理は実行されます。

「;（セミコロン）」を忘れずに記述します。

このプログラムでは、**do〜while**文の条件に「**sum < 0**」を指定し、合計を格納した変数**sum**が0よりも小さい間は**do〜while**文内の処理を繰り返します。変数**sum**の初期値は0であるため、初めから条件を満たしませんが、最初の1回は**do〜while**文内の処理が行われます。

このプログラムを実行した結果を次に示します。ソースコードの9～11行目の処理が一度だけ実行されていることを確認できます。

Column 制御文のネスト

制御構文は、別の制御構文の中に記述して入れ子にすることができます。これを制御構文の「ネスト」といいます。ネストを利用すると、「ループの中で条件判断を行う」といった複雑な処理を行うことができます。次にforの中でif文をネストしたプログラムを示します。

Sample0310.c　制御構文のネスト

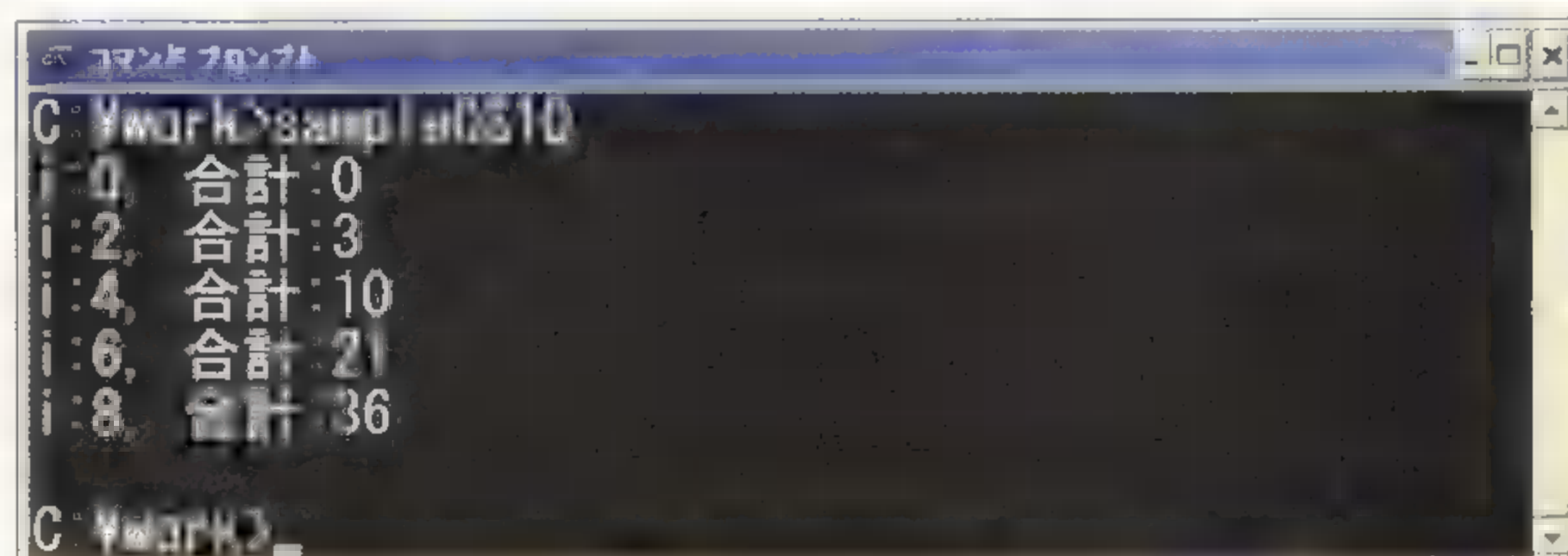
```
01  #include <stdio.h>
02
03  int main()
04  {
05      int i;
06      int sum = 0;
07
08      for (i=0; i<10; i++) {
09          sum += i;
10          if(i % 2 == 0) {
11              printf("i:%d, 合計:%d¥n", i, sum);
12          }
13      }
14      return 0;
15  }
```

ループ(for文)の中で、

条件判断(if文)を行います。

このプログラムでは、for文の中にif文をネストしています。for文では、0から9までの整数を順番に加算し、if文でループが偶数回の場合のみ画面表示を行います。

このプログラムを実行すると、偶数回のみ、画面に途中経過が表示されていることを確認できます。

```
C:¥work>sample0310
i:0, 合計:0
i:2, 合計:3
i:4, 合計:10
i:6, 合計:21
i:8, 合計:36

C:¥work>
```

4. break文

■ break文とは...

「**break**文」は、現在行っているループを終了し、ループから抜け出すための制御構文です。**switch**文でも利用することができます。**break**文はループの終了条件を詳細に指定したい場合に、**if**文などと組み合わせて使用します。

構文 break文

```
break;
```

次に示すのは「**break**文を利用して、**for**文によるループのカウンタが4以上になった場合に、ループを終了してループを抜ける」プログラムです。

Sample0311.c break文の利用

```
01 #include <stdio.h>
02
03 int main()
04 {
05     int i;
06     int sum = 0;
07
08     for (i=0; i<10; i++) {
09         sum += i;
10         if(i >= 4) {
11             break;
12         }
13         printf("i:%d, 合計:%d\n", i, sum);
14     }
15     printf("ループ処理から抜けました。\n");
16     return 0;
17 }
```

カウンタiが4以上の場合は、

ループを抜け出します。

5回目のループではカウンタ**i**が4になるため、11行目の**break**文が実行されます。**break**文が実行されるとその場でループが終了するため、5回目以降の画面出力（13行目）は行われません。これは実行結果からも確認できます。

```
C:¥work>sample0311
i:0, 合計:0
i:1, 合計:1
i:2, 合計:3
i:3, 合計:6
ループ処理から抜けました。

C:¥work>
```

5回目以降の画面出力が行われていません。

5. continue文

continue文とは...

「**continue**文」は、現在のループを中断して、次のループの頭から処理を継続する制御構文です。**continue**文も**break**文と同様に、ループの終了条件を詳細に指定したい場合に使用します。

構文　continue文

```
continue;
```

次に示すのは「**continue**文を利用して、**for**文によるループのカウンタが6以下の間は、画面出力の処理を飛ばしてループを継続する」プログラムです。

Sample0312.c　continue文の利用

```
01 #include <stdio.h>
02 
03 int main()
04 {
05     int i;
06     int sum = 0;
07 
08     printf("ループ処理を開始します。¥n");
09     for (i=0; i<10; i++) {
10         sum += i;
11         if(i <= 6) {
12             continue;
13         }
14         printf("i:%d, 合計:%d¥n", i, sum);
15     }
16     return 0;
17 }
```

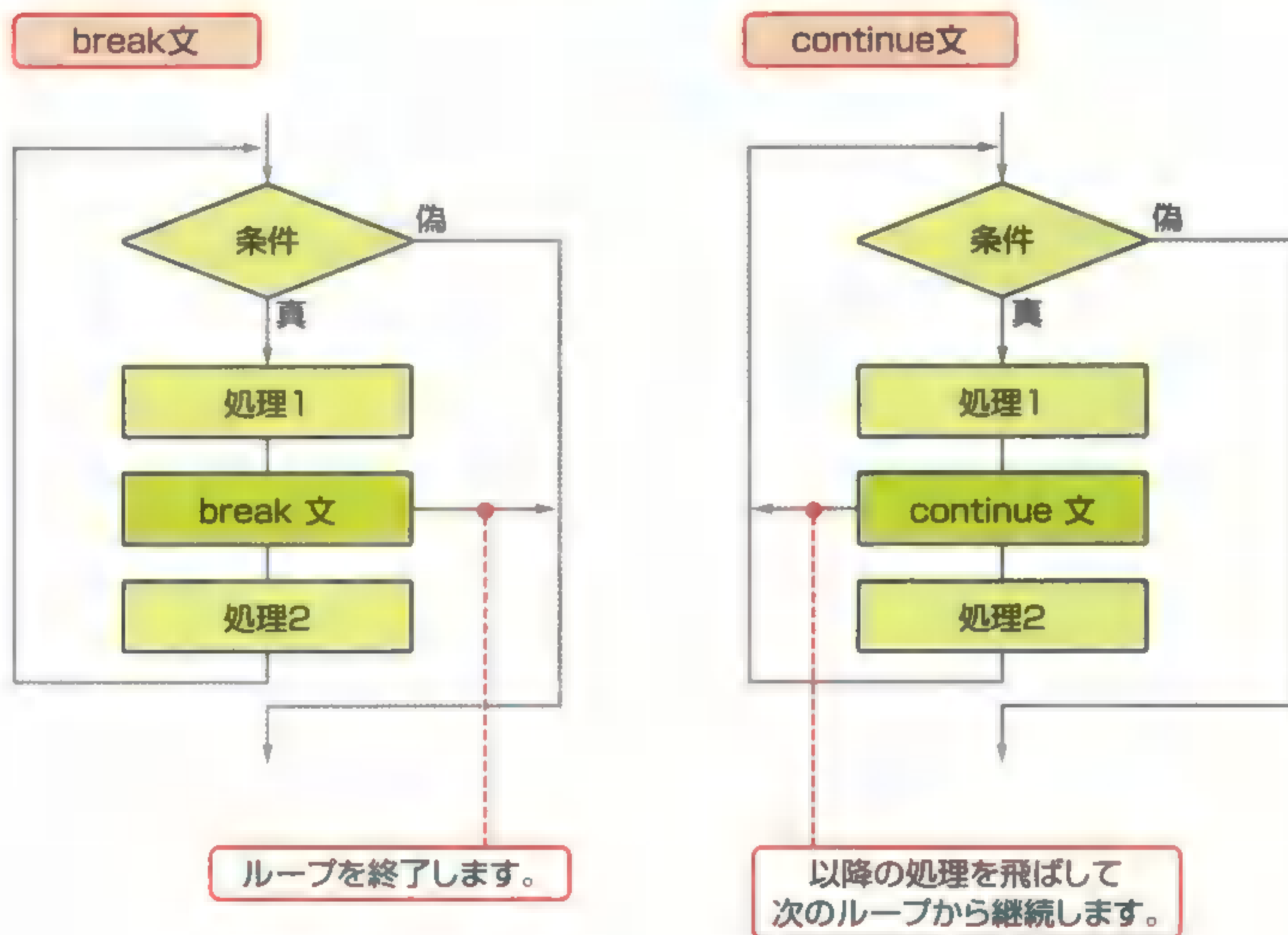
カウンタiが6以下の場合は、

以降の処理を飛ばして、次のループに移行します。

```
C:¥work>sample0312
ループ処理を開始します。
i:7, 合計:28
i:8, 合計:36
i:9, 合計:45

C:¥work>
```

実行結果から、**continue**文による処理の流れが確認できます。

break文と**continue**文の処理の違いをまとめると次のようになります。

図4 break文とcontinue文の違い

6. 無限ループ

■ 無限ループとは...

「無限ループ」とは、継続条件が常に真になり、半永久的に処理を繰り返して終わらないループのことです。たとえば、継続条件を省略した**for**文や、継続条件に値の「1」などを指定した**while**文などがあげられます。

for文を使った
無限ループ

```
for(;;) {
    …… 繰り返したい処理 ……
    if(ループを終了するための条件) {
        break;
    }
}
```

while文を使った
無限ループ

```
while(1) {
    …… 繰り返したい処理 ……
    if(ループを終了するための条件) {
        break;
    }
}
```


このような無限ループはユーザーからの入力を待つプログラムなどで使用されます。無限ループでは、**if**文や**break**文を組み合わせて、ループを終了するための処理を記述しておかなければなりません。もし間違えて、終了しない無限ループを含むプログラムを実行してしまった場合は、コマンドプロンプトで Ctrl + C を押すと、プログラムを強制的に終了することができます。

無限ループは、「ループが継続する条件」を記述するよりも、「ループが終了する条件」を記述した方がソースがわかりやすくなる場合などによく用いられます。

無限ループを利用したプログラムは、次のようになります。

```
int a = 0;
int b = 0;
int c = 0;
while( 1 ) {
    a += 3;
    b += 5;
    c += 7;
    if( (a >= 50) || (b >= 70) || (c >= 90) ) {
        break;
    }
}
```

無限ループは、「終了する条件」が
いくつもある場合などに利用されます。

まとめ 第3章: 制御構文

この章では、if文やswitch文など、処理の流れを条件によって分岐させる方法を学びました。また、for文やwhile文といった処理を繰り返す方法についても学習しました。これらの構文は、「上から順番に実行される」というプログラムの流れを制御するという意味で「制御構文」といいます。

第3章で学習したこと

- プログラムの流れは制御構文で変化させることができる。
- 制御構文は、条件の評価結果が「真」か「偽」かを判断し、処理を制御する。
- 制御構文には、大きく分けて「条件判断」と「ループ(繰り返し処理)」がある。
- ある条件を満たす場合の処理を記述するには、if文を利用する。条件を満たさない場合の処理を記述するには、else文を利用する。
- 条件判断に続いて複数の処理を連続して行いたい場合は、処理をブロックでまとめる。
- ある条件が、複数の条件のどれに該当するかを判断するには、switch文を利用する。
- 繰り返し処理にはfor文、while文を利用する。
- 繰り返し処理を抜けるにはbreak文を利用する。
- 一度繰り返し処理を抜けた後に再び繰り返し処理に戻るにはcontinue文を利用する。

ステップアップ!

この章の学習内容で、ようやくプログラミングらしいことができるようになりました。結局プログラミングは「どのように処理の流れを変えるか」を考え続ける作業です。たとえば、シューティングゲームであれば、「敵の弾が自機にあたったかどうか」、あたった場合には「まだ予備の機体は残っているか」など、条件分岐の連続です。どのようなプログラムを作る場合でも条件分岐はもっとも重要な要素であるため、この章で学習した条件分岐の方法はすべて頭に入れて状況に応じて適切に利用できるようになりましょう。

問1 条件に適合するかどうかの判断

次のプログラムの空欄を埋めてください。このプログラムでは変数**c**の文字コードが**'A'**か**'B'**かそれ以外かを判断します。

```
01 #include <stdio.h>
02 int main()
03 {
04     char c = 'A';
05     [    ] ( c ){
06         [    ] 'A':
07             printf("変数cの中身はAです。¥n");
08             [    ] ;
09         [    ] 'B':
10             printf("変数cの中身はBです。¥n");
11             [    ] ;
12         [    ] :
13             printf("変数cの中身はAでもBでもありません。¥n");
14             [    ] ;
15     }
16     return 0;
17 }
```

答1

いくつかある条件の中から適合する条件を探す場合は、**switch**文を利用します。条件と適合するかどうか比較する値の前には**case**を記述し、適合した場合に実行した処理から抜けるには**break**文を記述します。

```
01 #include <stdio.h>
02 int main()
03 {
04     char c = 'A';
05     switch ( c ){
06         case 'A':
07             printf("変数cの中身はAです。¥n");
```

```
08                break ;
09            case  'B':
10                printf("変数cの中身はBです。¥n");
11                break ;
12            default :
13                printf("変数cの中身はAでもBでもありません。¥n");
14                break ;
15        }
16        return 0;
17    }
```

問2 for文の利用

for文を用いて「変数**i**の初期値が5で、**i**が100になるまで処理を繰り返し、**i**は処理1回につき5増える」という制御構文を記述してください。

答2

for文では初期値、繰り返し条件、処理1回につき行う処理を記述できます。

```
for(i=5; i<100; i+=5 ) {
    ⋮
```

iは処理1回で5増えるので、i+=5と記述します。

問3 while文による無限ループ

条件式が常に真となるような**while**文を記述してください。また、その繰り返し処理の中で変数**i**をインクリメントし、**i**が100よりも大きくなったら繰り返し処理を抜けるようにしてください。

答3

条件式を常に真とするには、**while**文のカッコの中に0以外の数値を記述します。また、繰り返し処理を抜けるには**break**文を利用します。

```
int i = 0;
while( 1 ) {
    i++;
    if( i>100 ) {
        break;
    }
}
```


問4 if文

「変数**a**が50以下、なおかつ変数**b**が100以上だったら」という条件式を持つ**if**文を記述してください。

答4

「なおかつ」という条件を記述するには論理積の演算子(**&&**)を利用します。

```
if( (a<=50) && (b>=100) ) {
    ⋮
```

問5 do~while文

do~while文を用いて、10回繰り返すループ処理を行ってください。

答5

do〜while文は、条件のチェックを後で行う点だけが**while**文と違うところなので、条件式は**while**文と同様に、次のようになります。

```
int i = 0;
do {
    i++;
} while( i<10 );
```


問6 連続したif〜else文

if〜else文を利用して、変数の値が90以上か、80以上か、それ以外かで処理を分岐させてください。分岐した後に実行する処理には何も記述しなくてかまいません。

答6

if〜else文を続けて記述して、複数の条件に応じて処理を分岐させることができます。

```
int i = 85;
if(i >= 90) {
    ⋮
}
else if(i >= 80) {
    ⋮
}
else {
    ⋮
}
```

第4章

Visual Learning Introduction of C

配列

Section 13

配列の利用

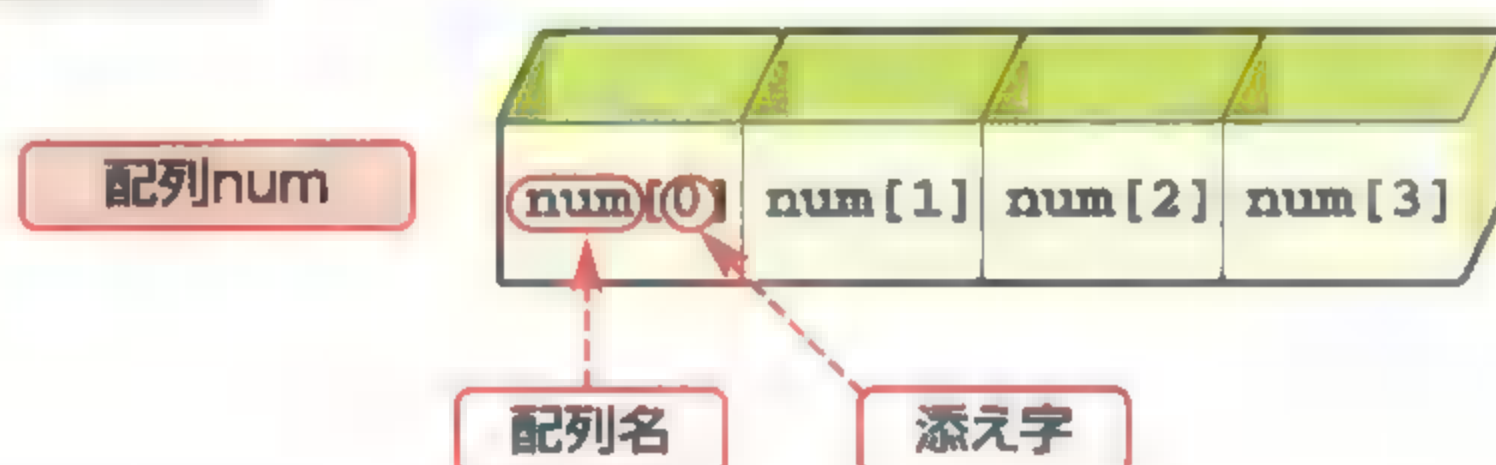
覚えておきたいキーワード
- 配列
- 添え字
- 配列の初期値

同じ型の複数のデータをまとめて管理するには、配列を利用すると便利です。また、文字列を表現するためには文字型変数の配列を利用します。配列の各要素は、配列名と番号（添え字）の組み合わせで表すことができます。

1. 配列の概要

■ 配列とは…

「**配列**」とは、複数の同じ型のデータをまとめて管理するしくみのことです。配列の各データのことを「**配列要素**」と呼び、配列名に「**添え字**」と呼ばれる連番を付けて区別します。

図1　配列

2. 配列の利用方法

■ 配列の宣言

配列を使う場合も通常の変数と同様に宣言が必要です。配列の宣言は次のように記述します。

構文　**配列の宣言**

```
データ型　配列名[配列の長さ];
```

データ型には「**int**」などの型を指定します。配列名は変数名と同様のルールでプログラマが設定します。配列の長さには整数の値を指定します。変数で長さを指定することはできません。

たとえば、「5つの要素を持った**int**型の配列**num**」を利用したい場合、次のように記述します。

```
int num[5];
```

■ 配列要素へのアクセスと代入

配列を宣言すると、配列名を利用して配列要素にアクセスすることができます。配列要素にアクセスするには、配列名と添え字を利用します。

配列要素へのアクセス

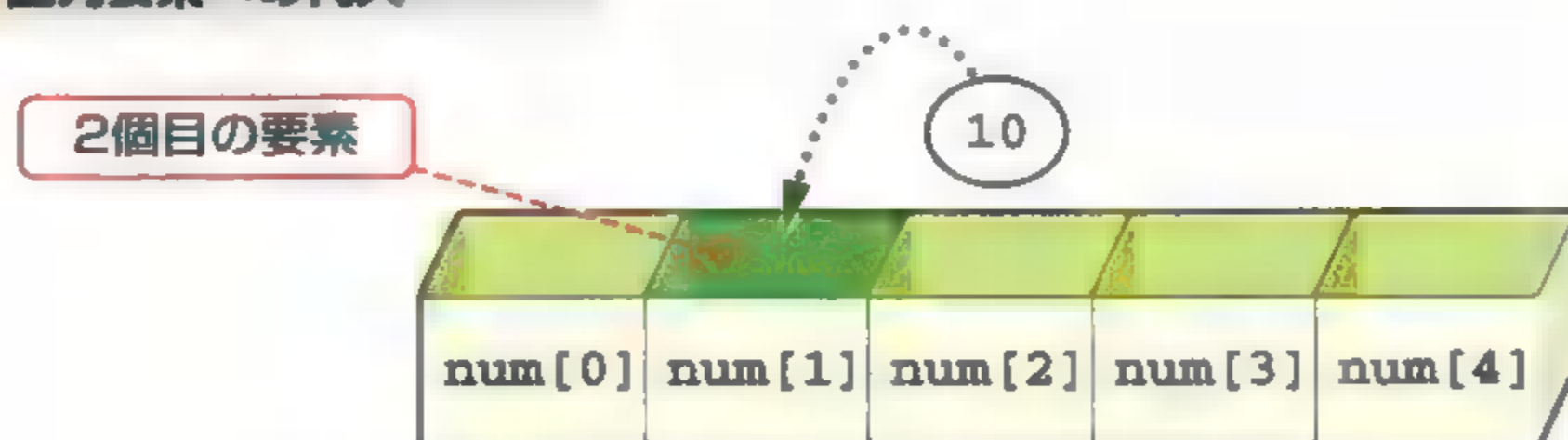
配列名[添え字]

添え字は配列の何番目の要素かを表し、0～(配列の要素数－1)の値が入ります。つまり、先頭の要素の添え字は0、末尾の要素の添え字は(配列の要素数－1)です。具体的には、5つの要素を持つ配列の添え字の範囲は、0～4となります。

たとえば、**int**型の配列**num**の先頭から2個目の要素に「10」を代入するには、次のように記述します。

```
num[1] = 10;
```

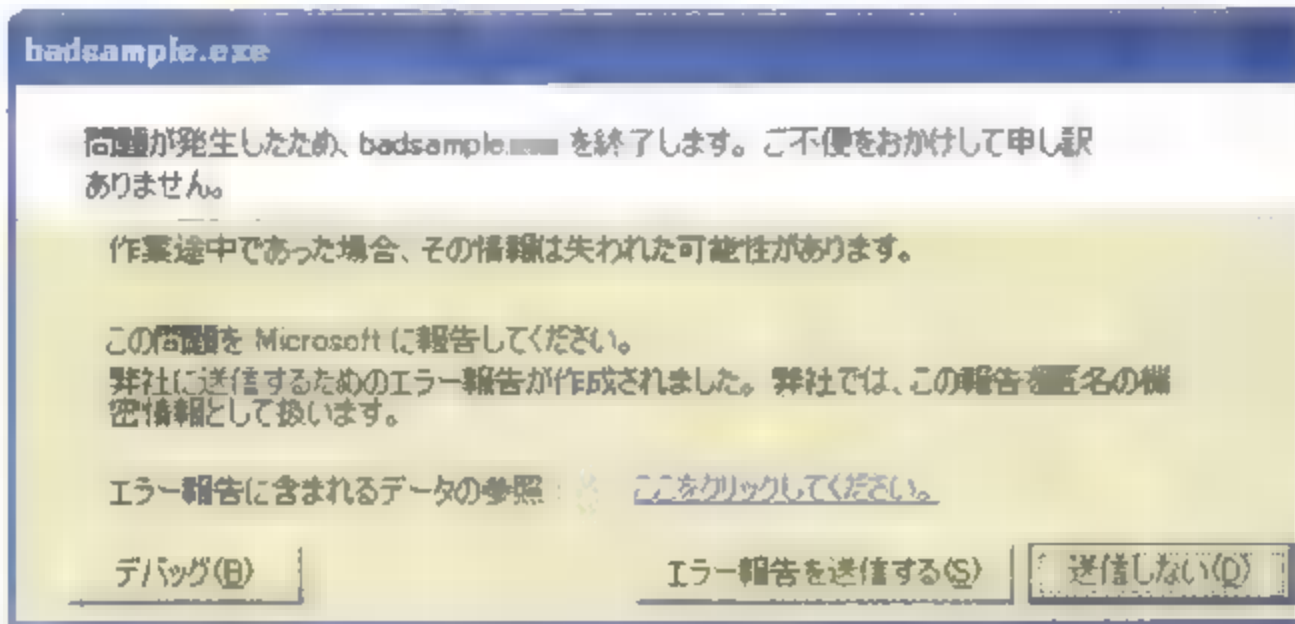
図2 配列要素への代入

2個目の要素
10
num[0] num[1] num[2] num[3] num[4]

「n個目」の要素を指定する場合、添え字は「n−1」になることに注意してください(前述の例でいうと、「2個目」の要素の添え字が「1」になります)。配列要素の添え字の値と「n個目」のnの値が異なると勘違いしやすくなるので、配列を使う際には一般的に先頭の要素を「0番目」として「0」から数え始めるようにします。

なお、C言語では、配列要素数以上の数値を添え字として記述しても、コンパイルエラーにはなりません。ただし、メモリ上で配列として確保されていない領域にアクセスしてしまうので、深刻なエラーになる可能性が高くなります。このように、配列の範囲外にアクセスしてしまうエラーのことを「**配列範囲外アクセス**」「**バッファオーバーフロー**」「**バッファオーバーラン**」などといいます。

これは、特に注意する必要のあるエラーのひとつです。

Windows XPでは、プログラムがバッファオーバーランなどにより、OSの利用するメモリ領域にアクセスしようとすると、以下のようなダイアログボックスを表示してプログラムを強制的に終了させせます。

図3 バッファオーバーランによるプログラムの強制終了

■ 効率的な配列の利用

種を明かすと、配列とは単に「同じ型の変数が並んでいるだけ」のものです。しかし、添え字によるアクセスを利用すると、次のように効率的なプログラムを作成できます。

```
int num[5];
int i;
int avg = 0;

num[0] = 65;
num[1] = 75;
num[2] = 70;
num[3] = 85;
num[4] = 80;

for(i=0; i<5; i++){
    printf("%d: %d¥n", i, num[i]);
    avg += num[i];
}

avg /= 5;
printf("平均:%d¥n", avg);
```

配列にデータを代入します。

配列のすべての要素の合計を求めています。

配列の各要素の平均を求めています。

```
C:¥work>sample0401
0: 65
1: 75
2: 70
3: 85
4: 80
平均:75

C:¥work>
```

この例では配列要素を累積して平均を求めています。添え字を変数で指定すれば、要素の値を累積する処理を`for`文で繰り返すことにより、簡単に記述できることに注意しましょう。このように配列と制御構文を組み合わせることで、データを効率よく利用することができます。

■ 配列の初期値の設定

変数と同様に、配列の宣言と同時に配列要素の初期値の設定を行うこともできます。

構文　配列の初期化

```
データ型 配列名[配列要素数] = { 初期値0, 初期値1, 初期値2, ……, 初期値n };
```

この書式を利用すると、先ほどのプログラムで、配列`num`への初期値の代入は、次のように記述することができます。

```
int num[5] = { 65, 75, 70, 85, 80 };
```

また、初期値を指定する場合には、配列要素数を省略して記述することも可能です。配列要素数を省略すると、{　}で囲まれた要素の数と等しい長さで配列が作られます。

構文　配列の初期化（配列要素数の省略）

```
データ型 配列名[] = { 初期値0, 初期値1, 初期値2, ……, 初期値n };
```

```
int num[] = { 65, 75, 70, 85, 80 };
```

このように配列の初期値を設定すると、{　}で囲まれた要素と同じ5つの配列要素を持つ配列が作られます。

Section 14

文字列

覚えておきたいキーワード
- 文字列
- 初期値
- 日本語の扱い

C言語では文字を扱うためのデータ型に「char型」がありますが、char型はわずか1文字の英数字を表すことしかできません。英単語など、2文字以上の英数字を利用したい場合や、漢字などの2バイト文字を扱いたい場合は「文字列」を利用する必要があります。

1. 文字列はchar型の配列である

今まではあえて「**文字列**」の説明を避けてきました。それは、文字列の説明がただ面倒であるからではなく、配列の概念への理解が必要であるためです。

C言語で文字列を扱うには、**char**型の配列を利用します。しくみは単純で、配列の先頭の要素から順番に文字を1文字ずつ格納していくだけです。ただし、文字列の共通の約束事として、「文字列が終了したら、最後の要素に値0を格納する」というルールがあります。

たとえば、「Hello」という文字列を利用したい場合、**char**型の配列に次のように格納します。

```
char str[6];

str[0] = 'H';
str[1] = 'e';
str[2] = 'l';
str[3] = 'l';
str[4] = 'o';
str[5] = '¥0';
```

文字列の最後には、必ず「¥0」を付けます。

「**'¥0'**」とは「NULL文字」ともいい、**エスケープシーケンス**のひとつです。実際には、値「0」と同じ意味ですが、文字列では「**0**」と「**'0'**」の混同を避けるなどの理由で「**'¥0'**」を利用するのが一般的です。

文字列用の配列を宣言するときには、文字列の末尾を表すNULL文字の分まで含めた長さ(文字列の長さ+1)を確保するように注意してください。

2. 文字列の代入

■ 初期値で文字列を代入する

char型の配列に文字列を一度に代入するには、配列を宣言すると同時に**初期値**を与える方法しかありません。文字列を初期値として利用するには、文字列を「"(ダブルクォーテーション)」で囲んだものを配列に代入します。

「"」で囲んだ文字列は、自動的に最後に「**'¥0'**」が付加されているとみなされます。つまり、その文字列を代入した配列の末尾にも「**'¥0'**」が付加されます。

配列の文字列による初期化

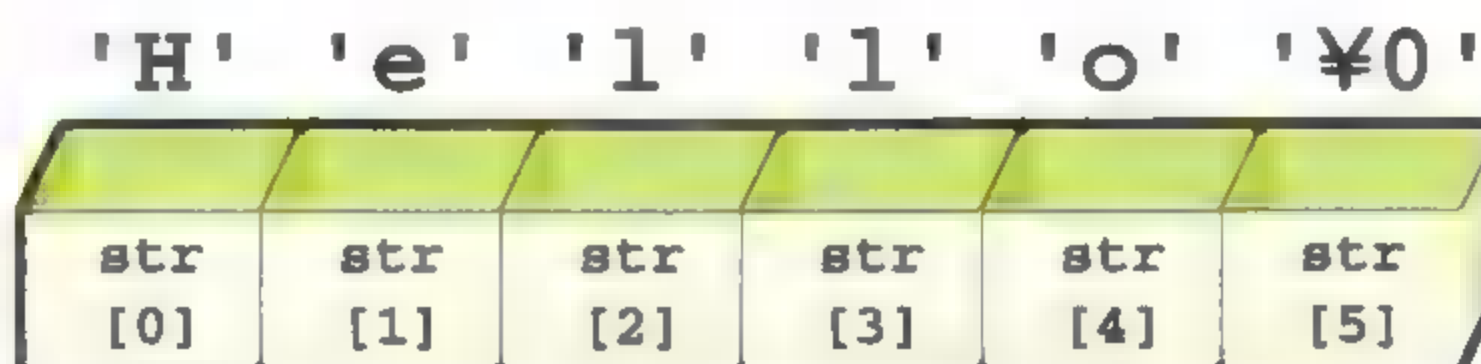
```
char 配列名[配列要素数] = "文字列";
```

通常の配列と同様に、初期値を指定する場合には配列要素数を省略することも可能です。

文字列の代入を利用したプログラムは、次のようになります。

```
char str1[] = "Hello";
char str2[6] = "Hello";
```

図1　文字列配列

文字列は、次のように配列に格納されます。

配列を文字列で初期化する場合、配列要素数は省略するのが一般的です。文字列の長さをわざわざ数えるのが面倒であったり、配列要素数に「**'¥0'**」の分を加算し忘れるのを防いだりするためです。

■ 配列に格納された文字列の変更

配列の宣言と同時に代入する以外の方法で、配列に文字列を設定することはできません。たとえば、次のように記述することはできません。

誤った例

```
char str1[] = "Hello";
str1[] = "Good morning";
```

文字列を修正する必要がある場合は、P.181の「**sprintf()**関数」を利用します。

3. 文字列の表示

文字列も**printf()**関数によって画面に表示することができます。文字列を画面に表示するには、変換指定子「**%s**」を利用します。

%sに文字列を指定するには、文字列が代入された配列の名前を記述します。

構文 printf()関数（文字列）

```
printf("%s", 配列名);
```

変換指定子（P.37参照）を利用したプログラム例を次に示します。

Sample0402.c 文字列配列

```
01 #include <stdio.h>
02
03 int main()
04 {
05     char str[] = "Hello";
06     printf("みなさん、%s¥n", str);
07     return 0;
08 }
```

文字列を表示する変換指定子です。

文字列を指定するには、配列名を記述します。

```
C:¥work>Sample0402
みなさん、Hello

C:¥work>
```

printf()関数を使って文字列を表示するには、変換指定子に**%c**ではなく「**%s**」を使うことと、配列名を記述することに十分に注意してください。

4. 日本語の取り扱い

配列に**日本語の文字列**を格納する場合も通常の文字列と同様に記述します。

```
char str[] = "おはよう";
```

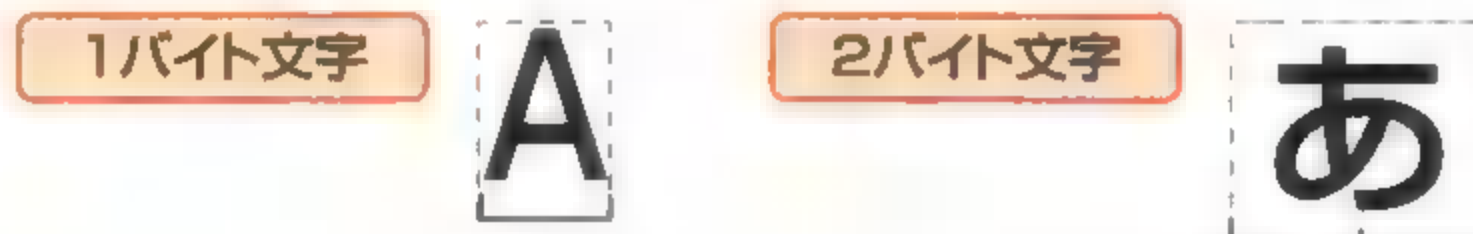
ただし、日本語のひらがなや漢字の表現は英数字に比べて特殊であるため、注意が必要です。

通常の英数字はよく「半角」と呼ばれます。半角は1文字を「1バイト」で表現します。**char**型はちょうど1バイトであるため、英数字を表現するのにちょうどよいサイズといえます。

これに対して、日本語はよく「全角」と呼ばれます。全角は1文字を「2バイト」で表現します。**char**型の変数では全角文字を表現できません。

図2　1バイト文字と2バイト文字

1バイト文字

2バイト文字

また、全角文字を「'(シングルクォーテーション)」で囲んでも、正しく文字コードを取得することはできません。加えて、2バイトの文字は**printf()**関数の変換指定子「**%c**」によって表示することもできません。

printf()関数で全角文字を扱う場合は、必ず「**%s**」で表示するようにします。

たとえば、次のようなプログラムは正しく動作しません。

誤った例

```
char c = 'あ';
```

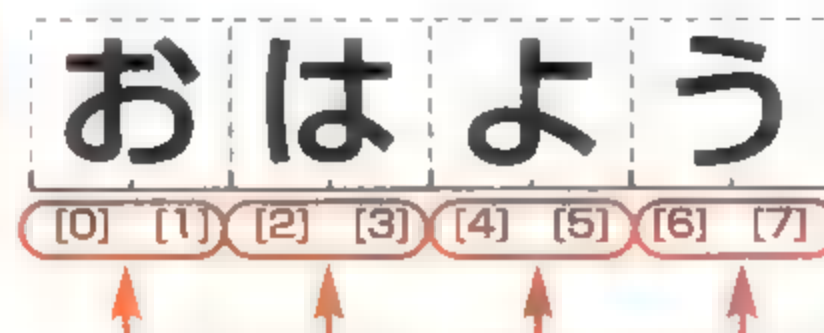
全角文字は「'」で文字コードを取得することはできません。

```
char str[] = "おはよう";
printf("%c", str[0]);
```

全角文字を%cで表示することはできません。

なお、全角文字の1文字を表現するためには2バイトが必要であるため、配列には次のように文字列が格納されます。

図3　全角文字の格納方法

2バイト文字の配列

char型配列の要素2つで全角文字1文字を表現します。

- 2次元配列
- 多次元配列
- 多次元配列の宣言

多次元配列

配列を利用すると同じ種類のデータをまとめて管理できます。人間が考えやすいように、たとえば「5×10の平面」のようにデータをまとめるのが多次元配列です。多次元配列を利用すると、プログラム内で表計算の表のようなデータを保持することができます。

1. 多次元配列の利用

配列の配列

これまでに解説してきた配列はすべてデータが1列に並んだ配列です。この配列を要素としてさらに配列を宣言することができます。このような「配列の配列」のことを「**多次元配列**」と呼びます。

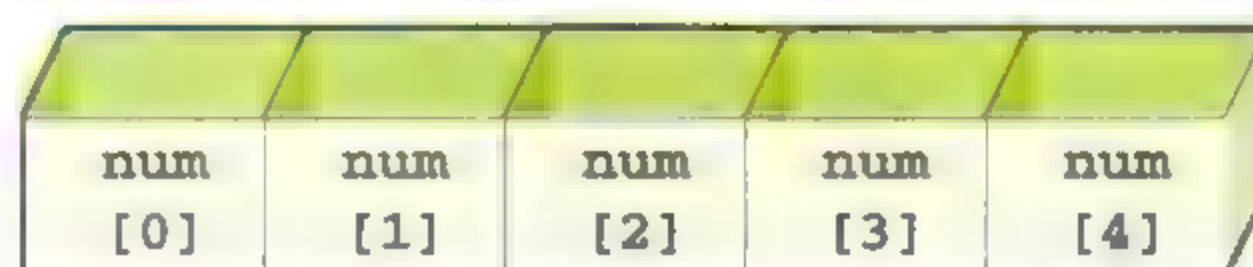
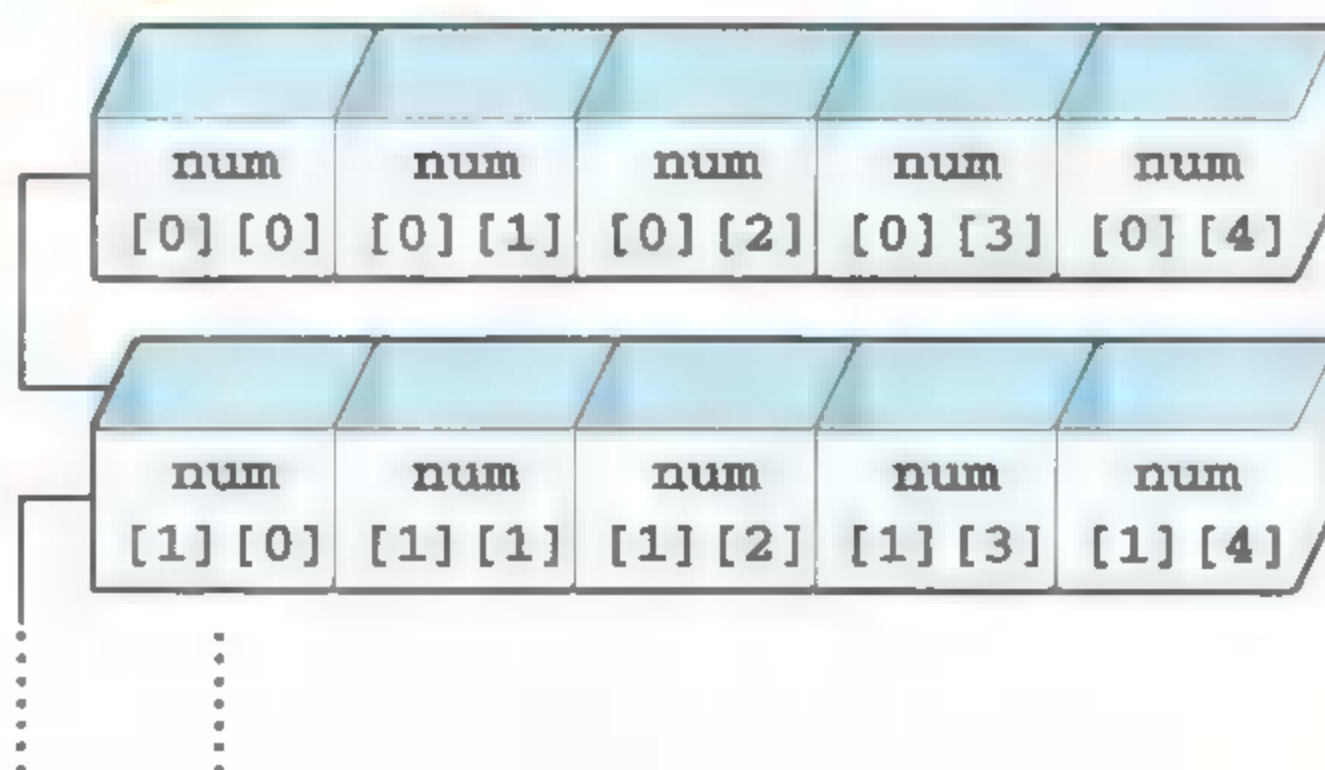



図1　1次元配列と2次元配列

また、「x軸」と「y軸」の2つの要素がある配列を「2次元配列」といいます。n個の要素がある配列は「n次元配列」と表現します。

■ 多次元配列の宣言

多次元配列は、次のように記述して宣言します。

構文　多次元配列の宣言

```
データ型 配列名[要素数][要素数]……;
```

また、配列の宣言と同時に初期化を行うこともできます。多次元配列を初期化するには、配列どうしを「,（カンマ）」で区切り、「{　}」を入れ子にして記述します。たとえば、2次元配列を初期化するには、次の書式に従います。

構文　2次元配列の初期化

```
データ型 配列名[要素数][要素数] = { {初期値0, 初期値1, …… },
                                    {初期値0, 初期値1, …… },
                                         ⋮
                                    {初期値0, 初期値1, …… } };
```

これらを利用したプログラムは、次のようになります。

Sample0403.c　多次元配列

```
01 #include <stdio.h>
02
03 int main()
04 {
05     int day, meal;
06     /* 7日×3食分の食費を保存する多次元配列 */
07     int food_exp[7][3] = {    {300, 850, 1000},
08                               {200, 700,  900},
09                               {200, 650, 1100},
10                               {250, 600, 1200},
11                               {300, 700, 2000},
12                               {250, 650,  950},
13                               {250, 700,  900} };
14     for(day=0; day<7; day++) {
15         printf("%d日目:");
16         for(meal=0; meal<3; meal++) {
```

2次元配列の初期化を行っています。

```
17            printf("%d円, ", food_exp[day][meal]);
18        }
19        printf("¥n");
20    }
21
22    return 0;
23 }
```

この添え字は、「7日」の要素を表します。

この添え字は、「3食」の要素を表します。

このプログラムでは、2次元配列の各要素を表示するために、for文を入れ子にして利用しています。この場合、3回繰り返す処理自体を7回繰り返しているので、3×7=21回処理を行います（下記参照）。

```
for(day=0; day<7; day++) {
    for(meal=0; meal<3; meal++) {
        処理;
    }
}
```

1 3回繰り返す処理を、

2 7回繰り返すので、

3 この処理は、3×7で21回行われます。

```
C:¥work>sample0403
0日目：300円, 850円, 1000円,
1日目：200円, 700円, 900円,
2日目：200円, 650円, 1100円,
3日目：250円, 600円, 1200円,
4日目：300円, 700円, 2000円,
5日目：250円, 650円, 950円,
6日目：250円, 700円, 900円,

C:¥work>
```

2次元配列の持つすべての要素（7日×3食）が、画面に表示されます。

また、文字列配列を多次元配列として扱うこともできます。これらを組み合わせたプログラムの例を次に示します。

Sample0404.c 文字列の多次元配列

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int day, meal;
06      /* 7日×3食分の食費を保存する多次元配列 */
07      int food_exp[7][3] = {    {300, 850, 1000},
08                                {200, 700,  900},
09                                {200, 650, 1100},
10                                {250, 600, 1200},
11                                {300, 700, 2000},
12                                {250, 650,  950},
13                                {250, 700,  900} };
14      /* 食事の種類を表示するための文字列の多次元配列 */
15      char meal_name[3][5] = {
16          "朝食", "昼食", "夕食"
17      };
18      for(day=0; day<7; day++) {
19          printf("%d日目:");
20          for(meal=0; meal<3; meal++) {
21              printf("%s%d円, ", meal_name[meal],
22                              food_exp[day][meal]);
23          }
24          printf("¥n");
25      }
26
27      return 0;
28  }
```

文字列の長さを表す要素です。

「3食」を表す要素です。

多次元配列の場合、配列名は
添え字を1つ減らしたものになります。

```
C:¥work>sample0404
0日目：朝食300円，昼食850円，夕食1000円，
1日目：朝食200円，昼食700円，夕食900円，
2日目：朝食200円，昼食650円，夕食1100円，
3日目：朝食250円，昼食600円，夕食1200円，
4日目：朝食300円，昼食700円，夕食2000円，
5日目：朝食250円，昼食650円，夕食950円，
6日目：朝食250円，昼食700円，夕食900円，

C:¥work>
```

まとめ 第4章：配列

この章では、変数をまとめて宣言する配列を学習しました。また、文字データ型(char型)の配列を利用して文字列を表現する方法も学びました。配列の配列ともいえる多次元配列の利用方法も学習しました。

第4章で学習したこと

- 配列とは、複数の変数をまとめて宣言する方法である。
- 配列では、配列名と添え字を使って変数にアクセスする。
- 配列の要素は、連続したメモリ領域に配置される。
- 配列を宣言するのと同時に初期値を設定することができる。この場合、配列宣言の添え字は省略することができる。
- 文字データ型の配列で文字列を表現し、文字列の末尾には必ず「'¥0'(NULL文字)」を追加する。
- 文字列を扱うには、文字列が格納された配列名だけを記述する。
- 日本語を表現したい場合は、必ず文字列を利用する。日本語は1文字で2バイト使うため。
- 多次元配列を利用すれば「配列の配列」が宣言できる。

ステップアップ！

配列は、ただ変数をまとめて宣言するだけではありません。本文中でも触れましたが、添え字によるアクセスが可能であるため、ループ処理で添え字に変数を使い、配列の中身をすべて0に設定したり、配列の中で値の大小を比較したりする処理を記述できます。配列は使い勝手がよく、配列の利点をうまく使いこなすようになれば、プログラミングの効率や処理速度を上げることができる重要な要素です。

また配列と同時に文字列の扱いも非常に重要です。C言語は文字列の扱いが不得意な言語であるといわれています。実際に記述してみると確かに少々面倒ではありますが、配列が理解できれば自然と文字列も理解できます。特に日本語を扱うときは必ず文字列として利用する必要があるので、しっかりと文字列の扱い方を覚えておくとよいでしょう。

問1　配列の宣言

int型の配列（要素数10）と、**float**型の配列（要素数5）を作成し、両方の配列のすべての要素を値「0」で初期化してください。

ヒント!　**float**型は浮動小数点数なので「0.0」で初期化する必要があります。

答1

配列の宣言はどのデータ型でも同じように記述できます。また、すべての要素を初期化するためにループ処理ですべての要素に同じ値を代入する方法を模範解答として示しましたが、要素と同じ数の初期値を代入するように配列を宣言する記述も正解です。

```
int i[10];
float f[5];
int n;
for(n=0; n<10; n++) {
    i[n] = 0;
}
for(n=0; n<5; n++) {
    f[n] = 0.0;
}
```

```
int i[10] = { 0,0,0,0,0,0,0,0,0,0 };
float f[5] = { 0.0, 0.0, 0.0, 0.0, 0.0 };
```

問2　文字列の表示

文字列「ただいま」と「おかえり」をそれぞれ別に**char**型の配列に格納し、**printf()**関数を使ってその**char**型配列を表示してください。その際、「ただいま」と「おかえり」の間に改行を入れてください。

答2

文字列を表示するには、**printf()**関数の変換指定子に「**%s**」を指定します。改行は「**¥n**」というエスケープシーケンスで表されるため、**%s**と**%s**の間に**¥n**を記述します。

```
char str1[] = "ただいま";
char str2[] = "おかえり";
printf("%s¥n%s", str1, str2);
```

問3 多次元配列

int型の5行×5列の2次元配列を宣言し、[0][0]に0、[0][1]に1、[0][2]に2、……、[4][4]に24を代入するプログラムを作成してください。ただし、ここでは宣言と同時に初期値を代入しない方法を考えてください。

答3

模範解答として、ループ処理で代入する方法を示します。

```
int i[5][5];
int j;
int k;
int sum = 0;
for(j=0; j<5; j++) {
    for(k=0; k<5; k++) {
        i[j][k] = sum;
        sum++;
    }
}
```

第5章

Visual Learning Introduction of C

関数

Section 16

関数とは

覚えておきたいキーワード
- 関数
- 引数
- 戻り値

ある決められた処理を何度も行いたい場合、関数を利用して処理の記述を1カ所にまとめることができます。処理を1カ所にまとめると、記述しやすくなるだけではなく、プログラムの修正やデバッグが容易になります。

1. 関数の定義

■ 関数とは…

「**関数**」とは、プログラム中に何度も登場する同じような処理を、1カ所にまとめて記述したもののことです。同じ処理を何度も記述するのに比べ、関数として1カ所にまとめることには、次に示すように大きなメリットがあります。

(1) 何度も同じコードを書かなくてよいので、手間がはぶける。

(2) どこで何を処理しているかを見つけやすい。

(3) 後からコードを修正するときに、修正箇所が1カ所ですむ。

(4) バグがあった場合、バグの発生箇所を特定しやすい。

なお、関数を利用するには「関数の定義」と「関数の呼び出し」という手順が必要です。続いて、これらの手順を順番に解説しましょう。

■ 関数の定義

関数を利用するには、まず、何らかの機能を実現するための一連の処理を関数として定義する必要があります。関数の定義は次のように記述します。

構文 **関数の定義**

```
戻り値の型　関数名(引数1の型　引数1,　引数2の型　引数2,　……)
{
    ……　関数で実行する処理　……
    return 戻り値;
}
```

今まで何度もプログラム例に登場した「**main()**関数」と同じ形であることに注目しましょう。異なるのは「引数」と「戻り値」をプログラマが自由に設定できる点です。

「**引数**」はこの関数が関数の呼び出し元から受け取るデータを示し、「**戻り値**」はこの関数が終了した後に関数の呼び出し元に返す値です。

■ 引数

関数が引数（ひきすう）を受け取るには、関数名の後の**()**の中に受け取る引数の型と名前を指定します。また、複数の引数を受け取る場合は、引数の型と名前を「**,**（カンマ）」で区切って列挙します。関数内では、受け取った引数を利用することが可能です。

```
int plus(int a, int b) {
    int c = a + b;
    ⋮
}
```

受け取った引数を利用します。

なお、引数を受け取らない関数を定義することもできます。その場合、引数の代わりに「**void**」と記述します。

```
int plus(void) {
```

引数を受け取らないことを示します。

■ 戻り値

関数の実行結果を1つの値として関数の呼び出し元に返すことができます。これを「**戻り値**」または「返り値」などといいます。戻り値は引数と異なり、1つしか返すことができません。

戻り値を返す場合には、あらかじめ関数の定義で「戻り値の型」を指定する必要があります。戻り値として返したい値は、関数内に「**return**文」を記述してそこに指定します。**return**文が実行されると関数の処理は終了し、戻り値が関数の呼び出し元に返されます。このとき、戻り値は、関数の定義に記述した「戻り値の型」のデータ型として呼び出し元に返されることに注意しましょう。

```
int plus(int a, int b) {
    return (a + b);
}
```

戻り値として呼び出し元に返されます。

なお、戻り値を返さない関数を定義することもできます。その場合、戻り値の型の代わりに「**void**」と記述し、**return**文を次のように記述するか、省略します。

```
void plus(int a, int b) {
    printf("%d", (a + b));
    return;
}
```

戻り値を返さないことを示します。
省略することもできます。

2. 関数の利用

■ 関数の呼び出し

では、実際に関数を利用してみましょう。定義した関数を利用するには、次のように記述します。

構文 関数の利用

```
関数名(引数1, 引数2, ……);
```

これは、**printf()**関数を利用する場合と同じ形であることに注意してください。**printf()**関数はC言語で最初から用意されている関数のひとつですが、関数の呼び出し方自体は同じです。

呼び出した関数から戻り値を受け取りたい場合は、あらかじめ戻り値を格納する変数を定義しておき、次のような形で関数を呼び出します。

構文 戻り値の利用

```
変数 = 関数名(引数1, 引数2, ……);
```

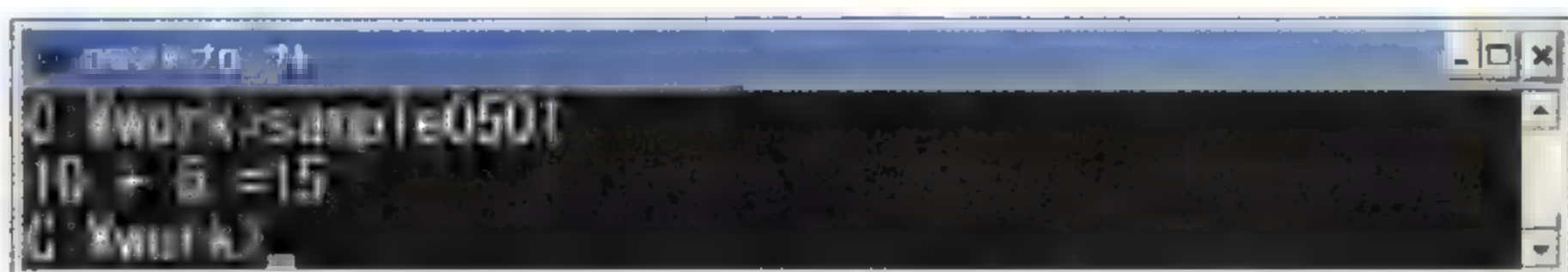
たとえば、**int**型の値を返す関数から戻り値を受け取るプログラムは、次のようになります。

Sample0501.c 関数から戻り値を受け取る

```
01  #include <stdio.h>
02
03  int plus(int a, int b)
04  {
05      printf("%d + %d =", a, b);
06      return (a + b);
```

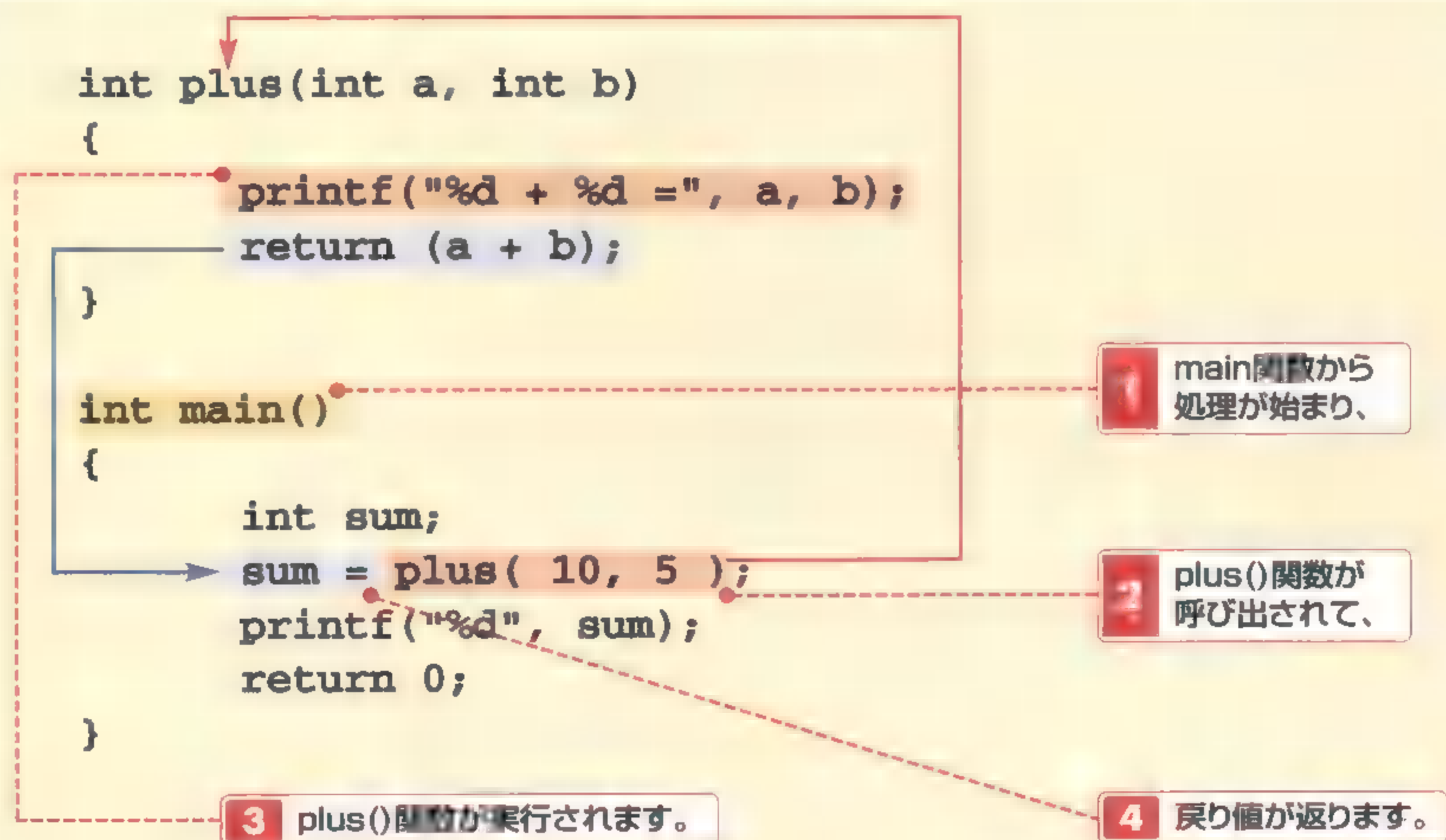
int型の戻り値を返す関数を定義します。

```
07  }
08
09  int main()
10  {
11      int sum;
12      sum = plus( 10, 5 );
13      printf("%d", sum);
14      return 0;
15  }
```

関数の戻り値をint型の変数に格納します。

この例のソースファイルでは、**main()**関数より上に**plus()**関数が記述されていますが、プログラムの処理は必ず**main()**関数から始まります。**main()**関数の中で**plus()**関数が呼び出されて初めて処理が**plus()**関数に移ります。

プログラムの実行順序を図にすると、次のようになります。

```
int plus(int a, int b)
{
    printf("%d + %d =", a, b);
    return (a + b);
}

int main()
{
        int sum;
        sum = plus( 10, 5 );
        printf("%d", sum);
        return 0;
}
```

1 main関数から処理が始まり、
2 plus()関数が呼び出されて、
3 plus()関数が実行されます。
4 戻り値が返ります。

なお、関数を呼び出すコードが書かれた行よりも上に関数が定義されていないと、エラーが表示されることがあります。そのような場合は、関数の呼び出しの記述より前に関数を定義しておくか、プロトタイプ宣言(P.116参照)を利用します。

Sample0502.c　関数の宣言位置(コンパイルエラー)

```
01  #include <stdio.h>
02
03  int main()
04  {
05      printf("こんにちは、");
06      weather();
07
08      return 0;
09  }
10
11  void weather(void)
12  {
13      printf("いい天気ですね。¥n");
14  }
```

関数の定義より前に関数の呼び出しが記述されています。

■ 引数の注意点

引数を受け取る関数に値を渡すと、その関数には、呼び出し元の値の**コピー**が渡されます。引数を受け取り、関数の中で受け取った引数の内容を変更しても、関数の呼び出し元の変数の値が変わるわけではありません。

Sample0503.c　コピー渡し

```
01 #include <stdio.h>
02
03 void plus(int data)
04 {
05     printf("関数の中> data = %d¥n", data);
06     data += 5;
07     printf("関数の中> data = %d¥n", data);
08 }
09
10 int main()
11 {
12     int num = 10;
13     printf("num = %d¥n", num);
14     plus(num);
15     printf("num = %d¥n", num);
16     return 0;
17 }
```

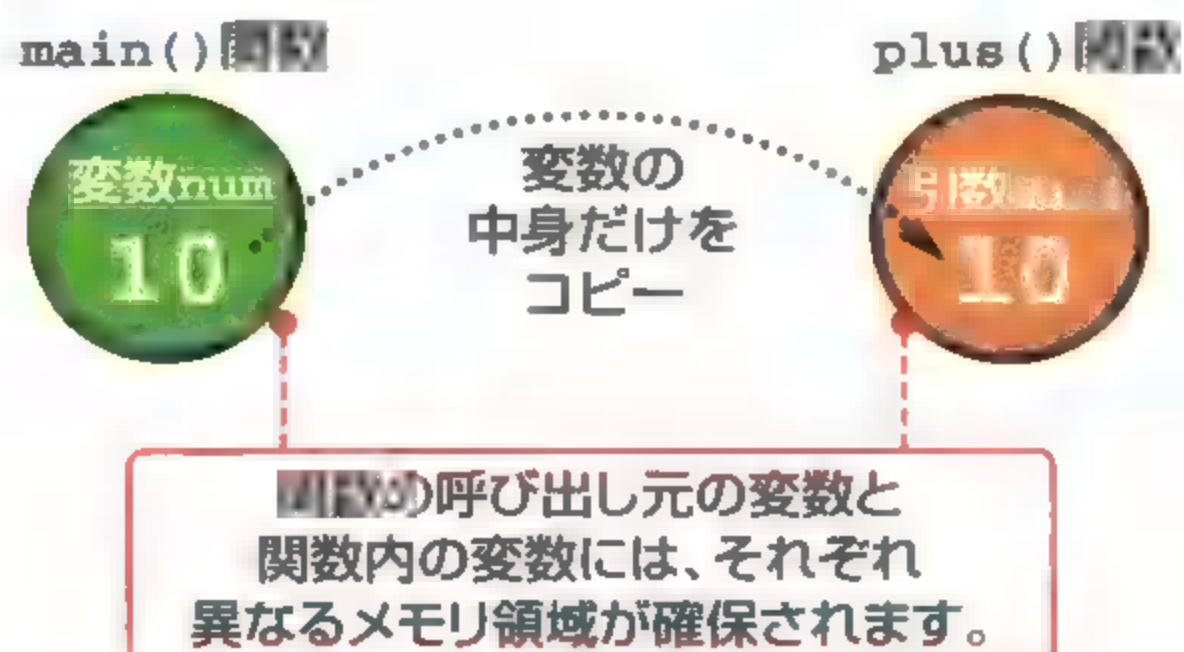
1 受け取った変数の値を変更したいが、

2 呼び出し元の変数の値は変化しません。

```
C:¥work>sample0503
num = 10
関数の中> data = 10
関数の中> data = 15
num = 10

C:¥work>
```

このように、関数の呼び出し元のコピーが渡されることを「**コピー渡し**」といいます。

図1 コピー渡し

覚えておきたいキーワード

- ローカル変数
- グローバル変数
- スコープ

ローカル変数とグローバル変数

関数の中で宣言した変数はローカル変数と呼ばれ、関数の外からアクセスすることはできません。関数の外に変数を宣言することもでき、これをグローバル変数と呼びます。グローバル変数は、ソースファイル内のどこからでもアクセスすることができます。

1. 関数の中でのみ有効な変数

関数の中で宣言された変数を「**ローカル変数**」といいます。ローカル変数は、その関数の中でのみ有効です。関数の外からアクセスすることはできません。

```
void foo(void)
{
    int num = 5;
    printf("%d", num);
}

int main()
{
    foo();
    num = 10;
    printf("%d", num);
    return 0;
}
```

1 ローカル変数numは、関数の中でだけ利用できます。

2 関数の外からは利用できません。(コンパイルエラー)

今までのプログラム例に記述してきた変数は、すべてローカル変数です。ローカル変数には、次のようなものがあります。

```
int main()
{
    int n;
}
```

関数の中で宣言される変数は、ローカル変数です。

```
int plus(int a, int b)
{
    ⋮
```

関数の引数も、ローカル変数です。

2. ソースファイル全体で有効な変数

一方で、同じソースファイル内であればどこからでも利用できる変数を作成することもできます。この変数を「**グローバル変数**」と呼びます。グローバル変数を作成するには、関数の外で変数を宣言します。

グローバル変数を利用したプログラムは、次のようになります。

Sample0504.c　グローバル変数

```
01  #include <stdio.h>
02
03  /* グローバル変数の宣言 */
04  int sum = 0;
05
06  void plus(int a, int b)
07  {
08      sum = a + b;
09  }
10
11  int main()
12  {
13      printf("sum = %d¥n", sum);
14      plus(10, 5);
15      printf("sum = %d¥n", sum);
16      return 0;
17  }
```

1 関数の中で、演算結果をグローバル変数に代入すると

2 関数の呼び出し元からも、変更された値を参照できます。

```
C:¥work>sample0504
sum = 0
sum = 15

C:¥work>
```

グローバル変数はたいへん便利な変数ですが、使いすぎるとどの変数がどこで使われているのかわからなくなり、バグの原因となります。そのため、必要最小限の利用ですむように注意しましょう。

3. スコープ

■ スコープとは…

「**スコープ**」とは、ひと言でいうとローカル変数の「寿命」です。スコープを外れると変数の寿命が終わり、宣言した変数は無効となります。**if**文などで作られる入れ子構造のブロックなどがスコープとなります。

スコープの先頭では、変数を宣言できます。関数の先頭部分で「**{**」に続けて変数を宣言しますが、それと同じ意味です。

入れ子構造の場合、スコープの外側で宣言した変数に内側からアクセスできますが、内側で宣言した変数に外側からアクセスすることはできません。


```
int i=0;

for(i=0; i<10; i++) {
    int a = 5;
    if(i > a) {
        break;
    }
}

if( a >= 2) {
    ⋮
}
```

1 この[illegible]は、

2 このスコープの終わりで無効となり、

3 スコープ外で利用しようとすると、コンパイルエラーになります。

4. 同じ名前の変数

すでに宣言しているローカル変数と同じ名前の変数を宣言しようとすると、コンパイルエラーになります。しかし、すでに宣言しているローカル変数と異なるスコープであれば、同じ名前の変数を宣言することができます。

その際、プログラムで扱えるのは、同じスコープで宣言された変数だけです。

Sample0505.c　別スコープの同じ変数名

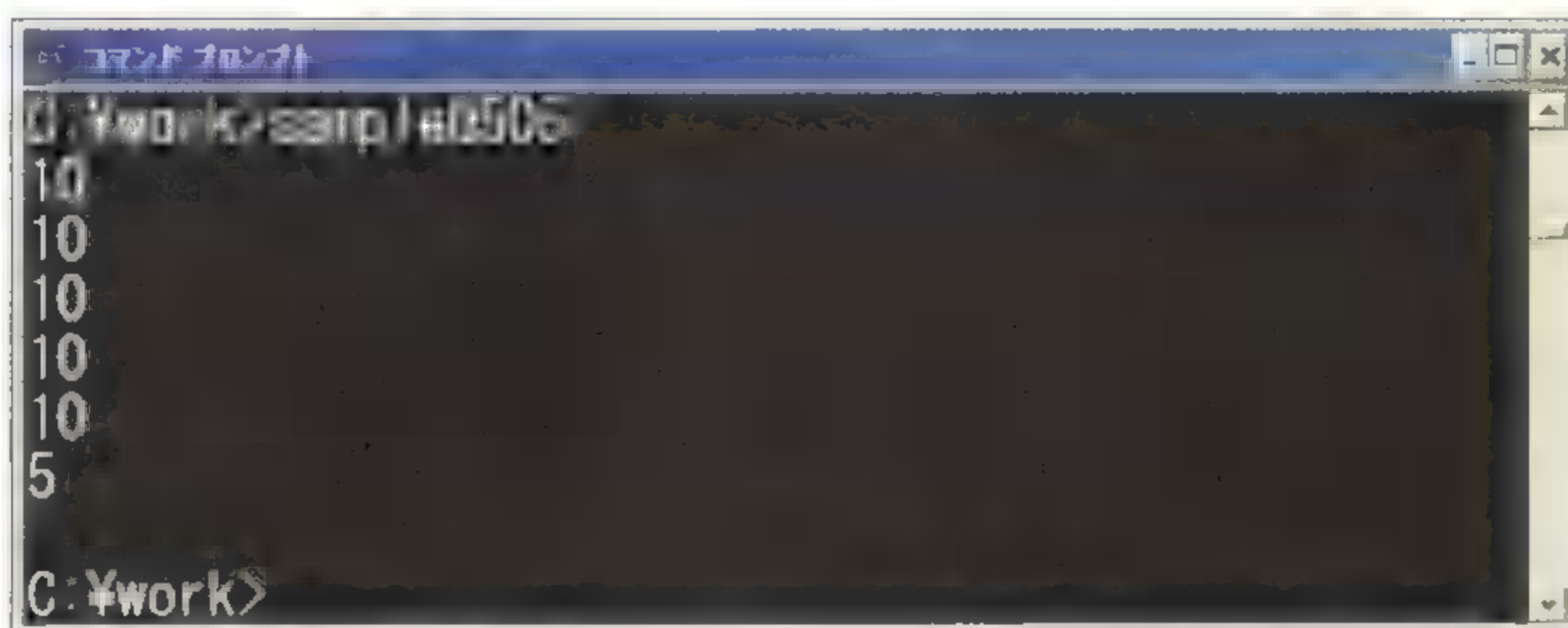
```
01  #include <stdio.h>
02
03  /* グローバル変数 */
04  int i = 100;
05
06  int main()
07  {
08      int i;
09      for(i=0; i<5; i++){
10          int i = 10;
11          printf("%d¥n", i);
12      }
13
14      printf("%d¥n", i);
15      return 0;
16  }
```

1 グローバル変数や、
2 このスコープで宣言した「i」ではなく、
3 同じブロック（for文のブロック）内で宣言した変数のみ扱えます。

このプログラムでは、**i**という名前の変数を3個宣言しています。11行目で変数**i**の内容を表示すると、10行目で宣言された**i**の内容が表示されます。これは、11行目で実行される**printf()**関数の呼び出しが10行目の**i**の宣言と同じスコープ、すなわち**for**文のブロック内に存在するためです。

一方、**for**文を抜けた後に**printf()**関数で**i**を表示すると、今度は8行目で宣言された**i**の内容が表示されます。14行目の**printf()**関数の呼び出しは、8行目の**i**の宣言と同じスコープ、すなわち**main()**関数のブロック内にあるからです。

このプログラムの場合、**main()**関数内からグローバル変数**i**にはアクセスできません。

```
C:¥work>sample0505
10
10
10
10
10
5

C:¥work>
```

Column static変数

ローカル変数は、処理が関数から呼び出し元に戻ると無効となります。関数が終了しても変数を有効なままにしたい場合は、「static変数」を利用します。static変数を利用するには、次のように記述します。

構文 static変数の宣言

```
static データ型 変数名;
```

見てわかるように、通常の変数宣言の先頭にstatic文を付けるだけです。static文を利用して宣言した変数は、処理が関数を抜けても有効なまま保持されます。ただしスコープはローカル変数と同じ扱いなので、関数の呼び出し元や他の関数からその変数を参照することはできません。

一度処理した関数をもう一度呼び出した場合、staticを付けて宣言した変数は、前回関数が呼び出されたときの値を保持しています。変数の宣言と同時に初期値を設定している場合、一番最初に関数が呼び出されときにだけ、その初期値は代入されます。2回目以降の呼び出しでは変数は初期化されません。

次のようなプログラムで、通常のローカル変数とstatic変数を利用して宣言した変数の違いがわかります。

Sample0506.c static変数の利用

```
01  #include <stdio.h>
02  
03  void foo(void)
04  {
05      int var = 10;
06      static int sta_var = 10;
07  
08      /* それぞれ変数を表示する */
09      printf("演算前¥n");
10      printf("staticなし:%d  staticあり:%d¥n",
11              var, sta_var);
12  
```

```
13      var += 50;
14      sta_var += 50;
15      printf("演算後¥n");
16      printf("staticなし:%d  staticあり:%d¥n",
17             var, sta_var);
18  }
19
20  int main()
21  {
22      printf("1回目の[illegible]び出し¥n");
23      foo();
24      printf("2回目の関数呼び出し¥n");
25      foo();
26      return 0;
27  }
```

関数の1回目の呼び出しでは、変数**var**も変数**sta_var**も値は変わりません。関数が2回目に呼び出されたときには、**sta_var**は前回関数を実行したときと同じ値から始まります。このように、**static**文を付けて変数を宣言すると、関数を抜けても変数の値が保持されるようになります。

Section 18

プロトタイプ宣言

覚えておきたいキーワード

- 関数の型
- コンパイラの構文解析
- プロトタイプ宣言

関数の呼び出しの前に関数を定義していないと、コンパイルエラーになってしまうことがあります。これはコンパイルの時点で、呼び出した関数の引数や戻り値の型がわからないためです。プロトタイプ宣言を利用すると、あらかじめこれらを定義しておくことができます。

1. 関数の型

関数にも「**型**」があります。関数の型は、「引数の型と数」と「戻り値の型」の組み合わせで考えます。たとえば、次の2つの関数は型が異なります。

引数の型が違う

```
int test1(int, char){
```

```
int test1(int, int){
```

戻り値の数が違う

```
int test2(int, int){
```

```
char test2(int, int){
```

コンパイラはコンパイルを行う際に「この関数の型はこれ」と認識しておき、その関数が呼び出されると関数の型に適応した方法で呼び出されているかをチェックします。

```
int plus(int a, int b)
{
    return (a + b);
}

int main()
{
    int n = plus( 10, 5 );
      ⋮
}
```

コンパイラは、plus( )関数の「型」を記憶します。

plus( )関数の型に応じた方法で呼び出されているかをチェックします。

2. プロトタイプ宣言の利用

コンパイラは、ソースファイルの解析を上から順番に行います。たとえば次のようなプログラムでは、関数の呼び出し元をコンパイルした時点で、その関数自体の型がわかりません。

```
int main()
{
    weather();
    return 0;
}
void weather(void)
{
    printf("今日はいい天気です。¥n");
}
```

ここをコンパイルした時点では、weather( )関数がどういう型なのかわかりません。

コンパイラはこのような場合、「とりあえず、**weather()**関数は戻り値が**int**型で、**int**型の引数を1個受け取る関数と仮定して処理を続けよう」と考え、コンパイルを続行します。

処理を続行すると**weather()**関数の定義が行われ、そこで初めて「さっきの仮定は間違いであった」と認識してエラーを表示し、コンパイルを中止します。

このような関数の型の不一致問題を避けるため、あらかじめ関数の型だけを宣言しておくことができます。これを「**プロトタイプ宣言**」といいます。

構文　プロトタイプ宣言

戻り値の型　関数名(引数1の型, 引数2の型, ……);

たとえば、前述の**weather()**関数のプロトタイプ宣言は、次のようになります。この宣言を**main()**関数より前に記述すれば、コンパイルは正しく行われます。

```
void weather(void);
```

プロトタイプ宣言を利用することで、関数を定義する位置を気にすることなくコーディングを行うことができます。また、ソースファイルを複数に分割してプログラムを作成するような場合には、プロトタイプ宣言を1カ所に集めておくことで、各ソースファイルにはどのような関数が含まれるのかを一覧で確認できるため、便利です。

再帰関数

C言語のプログラミングでは、関数を組み合わせて複雑な処理を行います。そのため、関数の利用についてはさまざまな応用法が用意されています。このセクションでは、関数を利用した代表的なテクニックとして、「再帰」を取り上げます。

1. 再帰関数とは...

関数を利用する際の代表的な応用例として「**再帰関数**」を説明しますが、関数の使い方を学習したばかりの状態ではやや複雑すぎるかもしれません。ここでは「再帰というものがあるんだ」という予備知識程度でとどめてもらってかまいません。

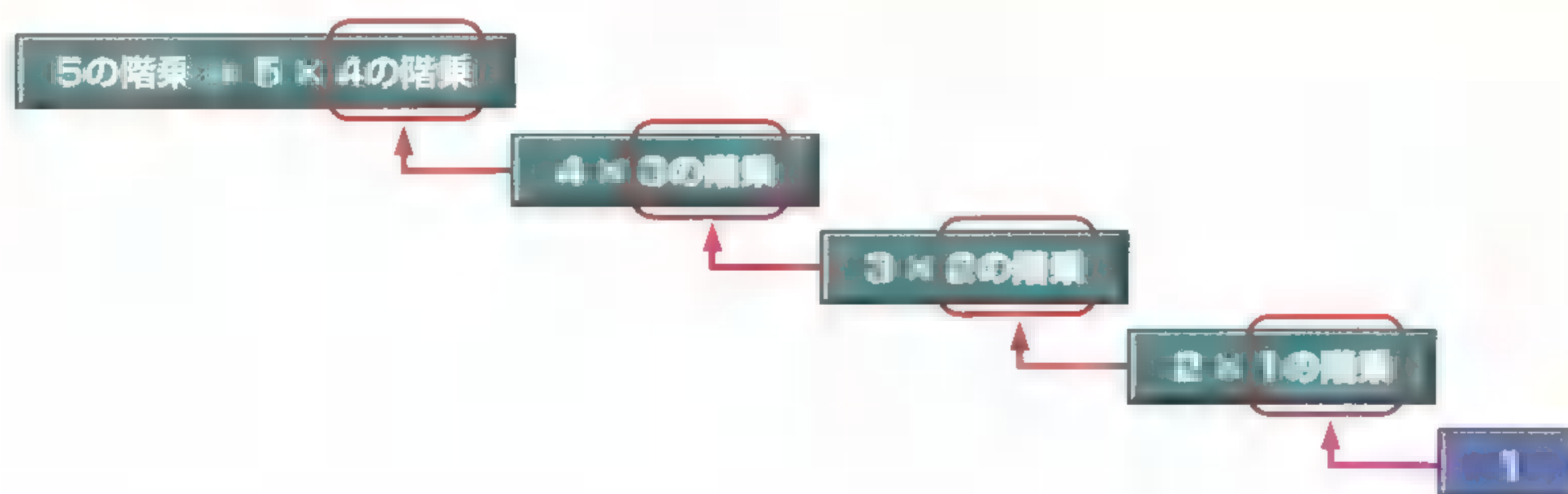
プログラムを作成する際に、「**関数の中で、その関数自身を呼び出す**」ということが必要になる場合があります。このような場合に「再帰関数」を使うと、複雑な処理を単純に実現できることがあります。ただし、すべてのプログラミング言語において再帰的な呼び出しが可能とは限りません。C言語は再帰関数を利用できる言語のひとつです。

2. 階乗の考え方

再帰の応用例として、よく階乗が取り上げられます。階乗とは、ある整数nから1までのすべての整数を掛けた数です。たとえば、5の階乗は5×4×3×2×1＝120と求められます。つまり、整数nの階乗を求める式はn×(n−1)×(n−2)×……×1です。

これを再帰関数を使って求めてみましょう。

まず、前述のとおり、5の階乗は5×4×3×2×1で求められます。ただし、よく見ると4×3×2×1は4の階乗です。つまり、5の階乗は5×4の階乗で求められます。同様に、4の階乗は4×3の階乗、……というように求められます。

図1　階乗の考え方

3. 階乗を求める再帰関数

　ここで、階乗を求める関数を**kaijo()**とし、引数は階乗を求める整数、戻り値は階乗の値とします。**kaijo()**関数を使って5の階乗を求めるには**kaijo(5)**と呼び出しますが、5の階乗は5×4の階乗で求められるため、**kaijo(5)**の呼び出しでは戻り値として**5 * kaijo(4)**を返すことができます。**kaijo(4)**や**kaijo(3)**も同様ですが、1の階乗は1であるため、**kaijo(1)**の場合のみ1を返します。

　この考えを再帰に適用すると、**kaijo()**関数を次のように作成できます。

```
int kaijo(int n)
{
    if(n>1) {
        return n * kaijo(n-1);
    }
    else {
        return 1;
    }
}
```

引数として渡された整数が1より大きい場合、(引数−1)を引数として自分自身を呼び出します。

引数が1の場合のみ、1を返します。

　このように再帰を使うと容易にプログラムを記述できる場合があります。ただし、再帰はあくまでも「やや高等なテクニック」です。無理に利用する必要はありません。

まとめ 第5章: 関数

この章では、関数の定義から呼び出しの方法までを学習しました。また、再帰関数など複雑なしくみにも簡単に触れました。関数を利用すると、ある決まった処理をまとめて記述でき、またプログラムの中から何度でもその関数を呼び出すことができます。

第5章で学習したこと

- あるまとまった処理を、関数としてまとめて記述することができる。
- 関数はプログラムの中で何度でも呼び出すことができる。
- 関数の中で宣言した変数をローカル変数という。この変数は関数の呼び出し元からは参照できない。
- 関数の引数もローカル変数のひとつである。
- 関数の引数は、関数の呼び出し元で指定した値のコピーを受け取る。
- ソースファイルのすべての位置から参照できる変数をグローバル変数という。
- 関数にも型があり、コンパイラに関数の型を認識させるために、関数の呼び出しの前にプロトタイプ宣言をする必要がある。
- 戻り値や引数を必要としない関数には、voidを指定する。

ステップアップ!

C言語は必ず処理がmain()関数から始まります。つまり、関数を組み合わせてプログラミングを行う言語だということができます。C言語プログラミングでは、1つの関数が無駄に長くなるのを避けて、処理を共通化できるところをうまく関数化していくことが一般的に行われます。

C言語でのプログラミングに慣れてくると、どのような処理を関数として定義できるかが徐々にわかってきます。うまく機能ごとに関数を作成し、できるだけ同じようなソースコードを作成しないようにすることが、C言語プログラミングのコツといえます。関数を利用したプログラミングに慣れて、効率的なプログラミングを習得しましょう。

問1 積の演算を行う関数

整数型の引数を2つ受け取り、その積を整数型で返す関数を作成してください。

答1

整数型として、引数と戻り値の型に**int**型を利用した解答例を次に示します。

```
int multi(int a, int b)
{
    return (a * b);
}
```

問2 ローカル変数

次のプログラムのグローバル変数を、すべてローカル変数に修正してください。

```
01  #include <stdio.h>
02
03  /* グローバル変数 */
04  int num1, num2;
05  int val1, val2;
06
07  void plus(void)
08  {
09      val1 = num1 + num2;
10  }
11
12  void multi(void)
13  {
14      val2 = num1 * num2;
15  }
16
17  int main()
18  {
19      num1 = 5;
20      num2 = 10;
```

```
21      plus();
22      printf("足し算: %d + %d = %d¥n", num1, num2, val1);
23
24      num1 = 3;
25      num2 = 5;
26      multi();
27      printf("掛け算: %d × %d = %d¥n", num1, num2, val2);
28
29      return 0;
30  }
```

答2

関数の引数もローカル変数になるように、関数を次のように修正します。

```
01  #include <stdio.h>
02
03  int plus(int a, int b)
04  {
05      return (a + b);
06  }
07
08  int multi(int a, int b)
09  {
10      return (a * b);
11  }
12
13  int main()
14  {
15      int num1, num2;
16      int val1, val2;
17      num1 = 5;
18      num2 = 10;
19      sum = plus(num1, num2);
20      printf("足し算: %d + %d = %d¥n", num1, num2, val1);
21
22      num1 = 3;
23      num2 = 5;
```

```
24      val2 = multi(num1, num2);
25      printf("掛け算: %d × %d = %d¥n", num1, num2, val2);
26
27      return 0;
28  }
```

問3 プロトタイプ宣言

次のプログラムは、プロトタイプ宣言がないためコンパイルエラーになります。プロトタイプ宣言を追加して、コンパイルが行われるように修正してください。

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int n = plus(5, 10);
06      printf("%d¥n", n);
07      return 0;
08  }
09
10  int plus(int a, int b)
11  {
12      return (a + b);
13  }
```

答3

ソースファイルの先頭部分に、次のようにプロトタイプ宣言を追加します。

```
01  #include <stdio.h>
02
03  int plus(int, int);
04
05  int main()
```

問4 引数を受け取らない関数

引数と戻り値を受け取らない**disp()**関数を作成し、その関数の中でグローバル変数**g**の値を表示してください。また、**main()**関数の先頭でグローバル変数**g**に値10を代入し、その後**disp()**関数を呼び出してください。

答4

引数や戻り値を利用しない場合には、データ型の代わりに**void**を記述します。

```
01  #include <stdio.h>
02
03  int g;
04
05  void disp(void)
06  {
07      printf("%d¥n", g);
08  }
09
10  int main()
11  {
12      g = 10;
13      disp();
14      return 0;
15  }
```

第6章

Visual Learning Introduction of C

ポインタ

Section 20

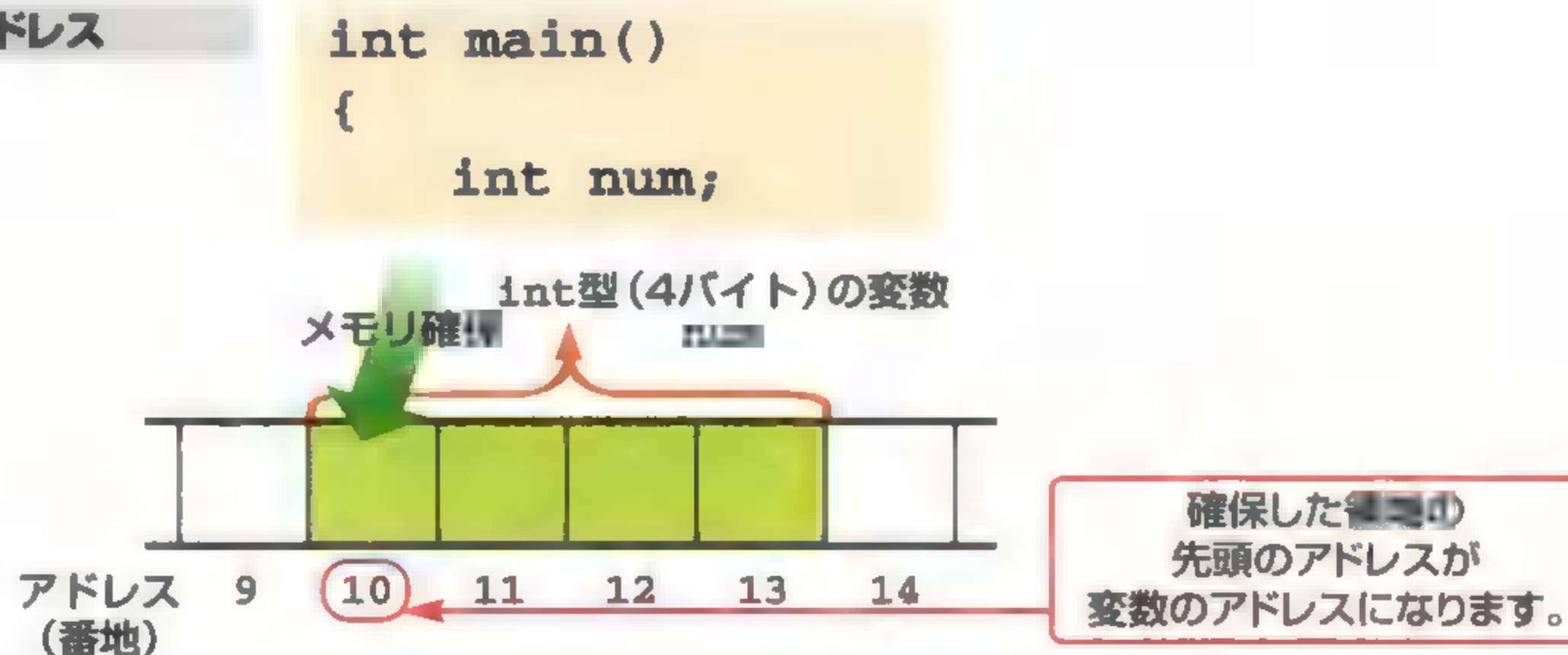
アドレスとポインタ

覚えておきたいキーワード
- アドレス
- ポインタ

プログラム内で変数を宣言すると、その変数の利用するバイト数に応じてメモリ領域が割り当てられます。変数が確保されたメモリ領域の場所を指し示すものを「アドレス」といいます。また、プログラム内でアドレスを利用するための変数を「ポインタ」といいます。

1. アドレスとは…

「**アドレス**」は、変数などが保存されている**メモリ**の場所を示すものです。アドレスはメモリの具体的な場所(「**メモリ番地**」ともいいます)を表します。

図1 変数とアドレス

変数のアドレスを表現するには、次のように記述します。

構文 アドレスの取得

```
&変数
```

「**&**」は、変数のアドレスを取得するための演算子で「アドレス演算子」と呼ばれます。たとえば、**int**型変数**num**のアドレスを表現するには、次のように記述します。

```
&num
```

配列の場合には注意が必要です。たとえば**int array[10]**という配列を宣言した場合、この配列の先頭のアドレスを表現するには、次のように記述します。

```
array
```

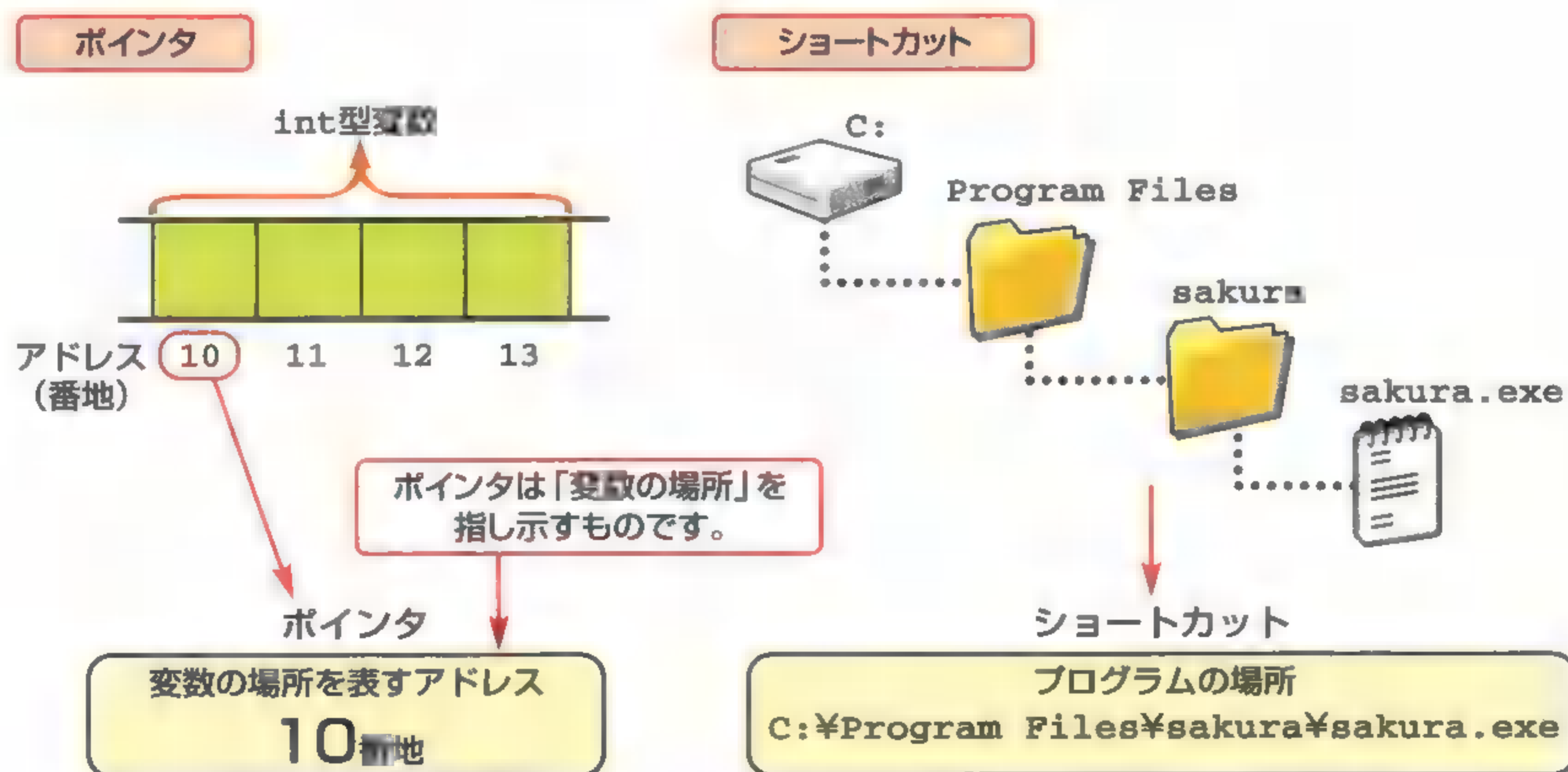
ただし、配列の特定の要素のアドレスを表現するには、変数と同様に次のように記述します。

```
&array[2]
```

2. ポインタとは...

■ ポインタの宣言

「**ポインタ**」とは、アドレスを保存するための変数のことをいいます。ポインタを利用するには、まずポインタに変数のアドレスを格納します。アドレスが格納されたポインタは「変数のメモリ領域の場所を指し示す」ものとして、通常の変数と同じような感覚で利用することができます。イメージとしてはWindowsのショートカットのようなものです。

図2　ポインタのイメージ

変数のアドレスを格納するためには、その変数と同じ型のポインタを宣言する必要があります。ポインタを宣言するには、次のように記述します。

構文 ポインタの宣言

```
データ型名 *ポインタ変数名;
```

たとえば、int型変数のポインタを宣言するには、次のようにします。

```
int *p;
```

■ アドレスの代入

ポインタには、変数のアドレスを代入できます。アドレスは変数の先頭に「&」を付けると取得できるので、ポインタにアドレスを代入するには次のように記述します。このとき、ポインタ変数とアドレスを代入する変数の型は同じであるものとします。

構文 アドレスの代入

```
ポインタ変数名 = &変数名;
```

また、ポインタの宣言と同時にアドレスを代入するには次のように記述します。

構文 アドレスの代入（宣言と同時）

```
データ型名 *ポインタ変数名 = &変数名;
```

■ ポインタの指す値

ポインタにはアドレスそのものが代入されており、そのままでは「そのアドレスに格納されている値」（つまり変数そのもの）を利用することができません。ポインタから、ポインタが指し示す変数を利用するには、ポインタを次のように記述します。

構文 ポインタが指す変数の利用

```
*ポインタ変数名
```

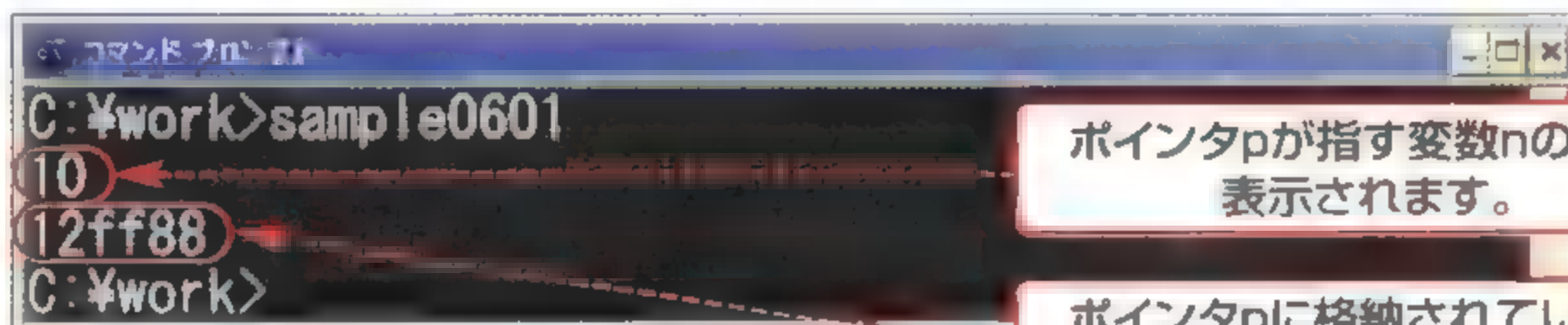
たとえば、変数のアドレスをポインタに格納し、ポインタを利用して変数の値を変更する場合、次のように記述します。

```
int n;
int *p;
p = &n;
*p = 5;
```

「p」に変数nのアドレスを格納します。

「*p」で変数nそのものを表します。

アドレスとポインタを利用したプログラムは、次のようになります。

Sample2001.c　ポインタの利用

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int n = 10;
06      int *p;
07      p = &n;
08      printf("%d¥n", *p);
09      printf("%x", p);
10
11      return 0;
12  }
```

pは、int型変数のポインタです。

ポインタpに、変数nのアドレスを代入します。

ポインタが指す場所に格納されている値にアクセスしたい場合は、「*p」と記述します。

16進数でポインタ変数自身の内容を表示すると、変数のメモリ番地が表示されます。

なお、ポインタは宣言しただけでそのまま利用することはできません。必ず変数のアドレスを代入してから利用するようにします。

ポインタの値は宣言したときには不定です。そのため、変数のアドレスを代入せずにポインタを利用すると、でたらめな値をアドレスとみなしてその場所を参照しようとして不特定のメモリ領域にアクセスしてしまいます。その結果、バッファオーバーラン（P.89）などと同様に、深刻なエラーの原因となる可能性がありますので、十分に注意が必要です。

Section 21

配列、文字列とアドレス

覚えておきたいキーワード

- 文字型配列
- 配列名のみの記述
- 配列とポインタの違い

C言語で文字列を扱うには、文字型(char型)の配列を利用します。まず、文字型の配列を確保し、先頭から順番に文字列を代入します。また、配列の先頭要素のアドレスを用いて「文字列全体」を表すこともできます。

1. 配列要素のアドレス

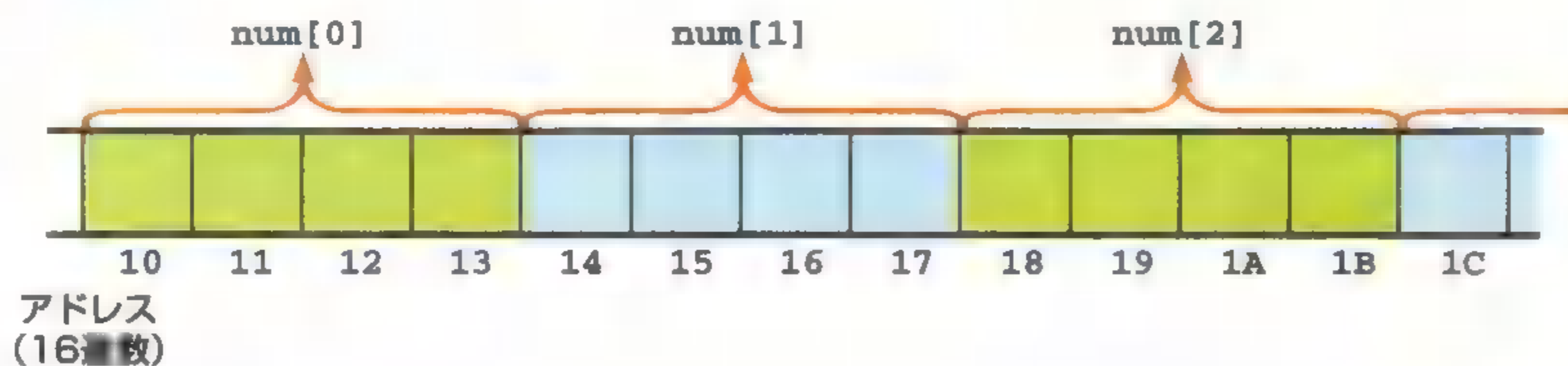
■ 配列要素ごとのアドレスの比較

Sec.13において、配列とは「複数の同じ型のデータをまとめて管理するしくみ」であると説明しました。配列を宣言すると、メモリ領域も1つにまとめられて確保されます。

たとえば、次のような配列がある場合を考えます。

```
int num[5] = { 1, 2, 3, 4, 5 };
```

この配列は、メモリ上に下図のように確保されます。

図1　int型の配列

では、これを確かめるために、次のようなプログラムを作成します。このプログラムでは、**int**型の配列を宣言し、各配列要素のアドレスを表示しています。

配列要素のアドレスは通常の変数と同様に「&」を付けて表現します。

Sample0602 配列要素のアドレス

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int num[5] = { 1, 2, 3, 4, 5 };
06      printf("[0]:%x, [1]:%x, [2]:%x, [3]:%x, [4]:%x",
07              &num[0], &num[1], &num[2], &num[3], &num[4]);
08      return 0;
09  }
```

要素に「&」を付けてアドレスを表現します。

```
C:\work>sample0602
[0]:12ff78, [1]:12ff7c, [2]:12ff80, [3]:12ff84, [4
]:12ff88
C:\work>
```

値はすべて16進数で表示しています。ちなみに、アドレス情報を表示する場合、16進数で行うのが一般的です。

実行結果の画面表示より、配列はつながったメモリ上にまとめて確保されるのがわかります。

2. 文字列の表現のしくみ

■ 配列名のみの記述

最初に、**printf()**関数で文字列を画面に表示するときのことを思い出してください。

```
char str[] = "こんにちは";
printf("%s", str);
```

この例では、**char**型配列の「**str**」に文字列を代入し、それを**printf()**関数を利用して表示しています。では、**printf()**関数のカッコの中に記述した「**str**」とは、何を指すのでしょうか?

実は「**str**」は、「**str**という配列の0番目の要素のアドレス」を表しています。つまり、前述の例は、次のようにも記述することができます。

```
char str[] = "こんにちは";
printf("%s", &str[0]);
```

C言語では、いちいち「**&str[0]**」と記述するのが面倒であるため、「**str**」と書いたら配列そのもの、つまり配列の先頭要素のアドレスを示すことにしようと定められています。

「**str**」と「**&str[0]**」が同じアドレスを指していることを、その内容を画面に表示して確かめてみましょう。

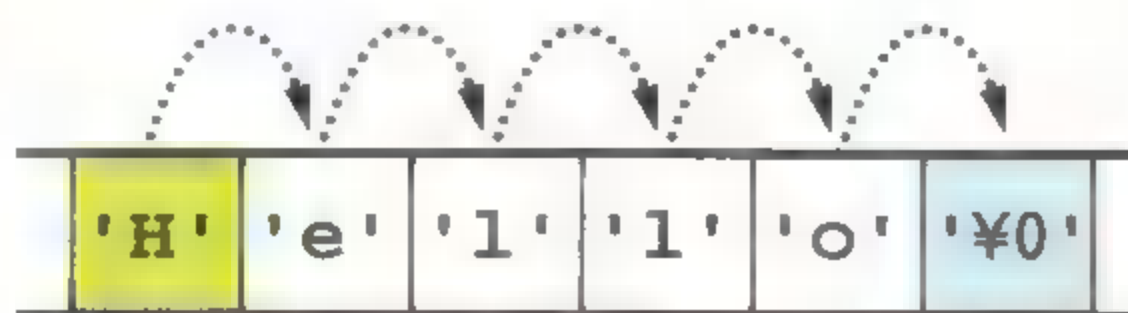
```
char str[] = "こんにちは";
printf("&str[0]:%x, str:%x", &str[0], str);
```

このように、同じアドレスを指していることがわかります。

■ 文字列の終端

Sec.14で文字列には終端に「**'¥0'**」を入れる必要があると説明しました。C言語では複数の文字が並んで格納され、最後に**'¥0'**が格納されている文字型配列を**文字列**とみなします。

また、**printf()**関数など、文字列を扱う関数には「先頭の要素から順番に**'¥0'**が保存されている要素までを表示する」というように**'¥0'**を文字列の最後を示す目印として処理を行うものが多く存在します。このような関数では、配列がどれだけの長さで確保されているかなどの条件を考慮しないで処理を行うため、「**'¥0'**」がないと配列の長さを超えてバッファオーバーラン（P.89参照）を引き起こします。

図2 文字列の終端

受け取ったアドレスから順番に
'¥0'を探してアクセスします。

3. 配列とポインタの違い

配列とポインタでは、次のような違いがあります。

表1　配列とポインタの違い

配　列	ポインタ
宣言した時点でメモリ領域が確保されています。	宣言するだけではメモリ領域は確保されません。
配列名の指すアドレスを後から変更できません。	ポインタが指すアドレスを後から変更できます。

なお、メモリ領域については、ポインタでも次のように記述すると宣言と同時にメモリ領域を確保することができます。

```
char *str = "こんにちは";
```

また、配列と同様にポインタでも、添え字による各要素へのアクセスが可能です。たとえば、次のように記述できます。

```
char str1[] = "hello";
char *str2 = str1;
printf("%c", str2[1]);
```

「str1」で文字列の先頭アドレスが表現されるので、「&」は必要ありません。

str2はchar型のポインタですが、配列のように添え字による各要素へのアクセスが可能です。

printf()関数など、文字列を引数として受け取る関数では、配列を受け取ると配列名が指すアドレスを後から変更できず、不便であることから、文字列を「**char**型のポインタ」として受け取って利用します。配列とポインタはデータ型が同じであれば実質的にほぼ同じように利用できるものなので、配列名をポインタとして利用したり、ポインタと添え字で配列のように利用したりすることができます。

Section 22

覚えておきたいキーワード
- ポインタの引数
- 参照渡し
- 配列の引数

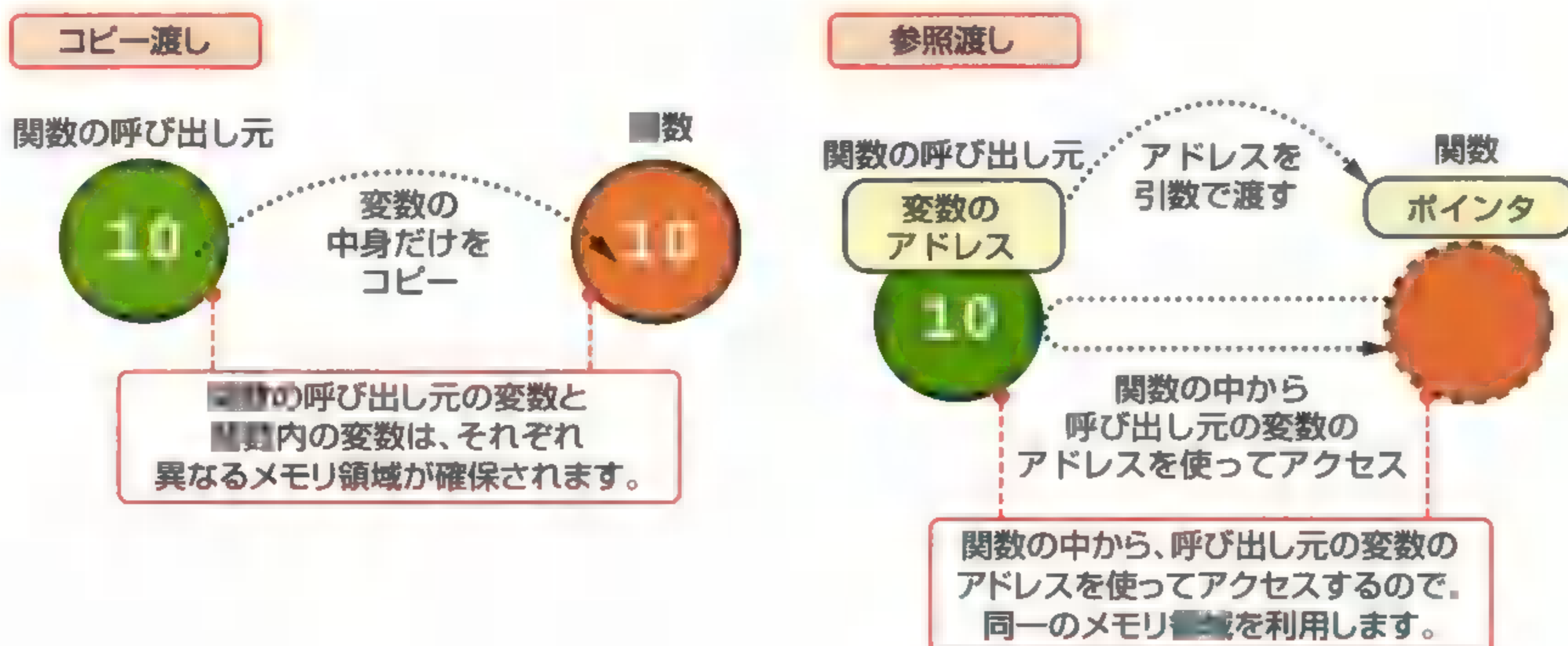
ポインタを受け取る関数

ポインタを利用すると、関数の呼び出し元の変数の内容を関数の中で変更することができます。呼び出し元の変数のポインタを受け取ることで、戻り値を使わずに実行結果を返せるため、複数の値を返したい場合などに利用できます。

1. ポインタを引数に利用する利点

引数としてポインタを渡す

関数では、引数にポインタを利用することができます。引数に変数を利用する場合、関数の呼び出し元から渡された値が引数に代入されますが、引数にポインタを利用するとポインタの内容、すなわち特定のメモリ領域へのアドレスが引数に代入されます。つまり、関数の呼び出し元と関数の中において、同じメモリ領域を参照できるということです。

図1　引数のコピー渡しと参照渡し

ポインタを引数として利用すると、関数の中からそのポインタを利用して関数の呼び出し元で宣言した変数の値を変更することができます。

ポインタの引数を利用したプログラムは、次のようになります。

6 ポインタ

Sample0604.c ポインタの引数

```
01  #include <stdio.h>
02
03  void plus(int *p)
04  {
05      printf("plus()関数の引数pのアドレス:%x¥n", p);
06      printf("plus()関数の引数pの値:%d¥n", *p);
07      *p += 10;
08      printf("plus()関数の引数pの値:%d¥n", *p);
09  }
10
11  int main()
12  {
13      int num = 10;
14
15      printf("変数numのアドレス:%x¥n", &num);
16      printf("変数numの値:%d¥n", num);
17      plus(&num);
18      printf("変数numの値:%d¥n", num);
19
20      return 0;
21  }
```

- ポインタを引数として受け取ります。(03行目)
- ポインタが指す呼び出し元の変数の内容を変更します。(07行目)
- 変数のアドレスを引数として渡します。(17行目)

このプログラムを実行すると、結果は次のようになります。

```
コマンド プロンプト
C:¥work>sample0604
変数numのアドレス:12ff88
変数numの値:10
plus()関数の引数pのアドレス:12ff88
plus()関数の引数pの値:10
plus()関数の引数pの値:20
変数numの値:20

C:¥work>
```

- 変数の値が変更されています。(変数numの値:20)

17行目で**plus()**関数を呼び出していますが、このときに**main()**関数で宣言した変数**num**のアドレスを引数として渡しています。これにより、変数**num**のアドレスを**plus()**関数では「**int *p**」というポインタとして利用することができます。

ここでは変数**num**のアドレスと、**plus()**関数で受け取った引数**p**のポインタが指すアドレスが同じであることに注目してください。これにより、7行目の「***p += 10;**」の演算が、変数**num**に適用されることになります。その結果、**plus()**関数から処理が戻った後に変数**num**を表示すると値が10増えていることがわかります。

■ 引数として配列を渡す

関数の引数として、配列を渡すこともできます。引数として配列を指定する場合には、要素数を記述しないようにします。たとえば、配列を受け取る関数は、次のように記述します。

```
void foo(int num[])
{
    ⋮
```

また、引数としてポインタを受け取る関数に、配列を渡すこともできます（P.133参照）。

```
void foo(int *num)
{
    ⋮
```

いずれの場合も関数側では配列の先頭要素のアドレスを受け取り、そのアドレスにより配列の要素にアクセスします。関数の中で配列の各要素の値を変更すると、関数を抜けて呼び出し元に戻っても、値の変更が反映されています。

このように、関数の引数に配列のアドレスを利用すれば、配列の要素すべてをわざわざコピーしなくても関数の中で配列を参照することができ、便利です。

Column ポインタの演算

Sec.21では、ポインタと添え字を使って配列にアクセスする方法を解説しました。たとえば、右のコードを実行すると、文字「W」が表示されます。

```
char str[] = "Hello World";
char *strp = str;
printf("%c¥n", strp[6]);
```

一方で、添え字を使わずにポインタを右のように使用しても同様の処理が可能です。

```
printf("%c¥n", *(strp+6));
```

この例ではポインタが参照する位置を6文字分先に進めてから、ポインタが指す値を取得しています。このように、ポインタに整数値を足す、または引くことによって、ポインタが参照する位置を変えることができます。
また、ポインタに対してインクリメント演算子やデクリメント演算子を適用することも可能です。

```
char str[] = "Hello World";
char *strp = str;
int i;
for(i=0; i<11; i++) {
    printf("%c¥n", *strp++);
}
```

ポインタが参照する値を使用してから
ポインタをインクリメントします。

このプログラム例を実行すると配列strに格納されている文字が1行に1文字ずつ画面に表示されます。配列strを参照するstrpをインクリメントすると、ポインタstrpは1文字ずつ先に進んだ値を参照するようになります。なお、ここでは後置型のインクリメント演算子を使っているため、*strpで文字を参照してからstrpの参照位置を1文字分インクリメントしていることに注意しましょう。

このようなポインタの演算では、前述の例のように6を足したり、インクリメント演算子により1を足したりしても、ポインタが参照する位置が必ず6バイト分あるいは1バイト分先に進むわけではありません。ポインタの演算により進む距離は、ポインタのデータ型に依存します。
では、int型のポインタintpを使う場合を考えます。

```
int ints[] = {10, 20, 30, 40, 50}";
int *intp = ints;
int i;
for(i=0; i<5; i++) {
    printf("%d¥n", *intp++);
}
```

int型のポインタをインクリメントすると
ポインタが参照する先が
4バイトずつ進みます。

このプログラムではint型のポインタを使ってint型の配列の内容を参照しています。ポインタintpをインクリメントすると、配列の次の要素、つまり4バイト先を指し示すようになり、配列要素が1つずつ表示されます。ポインタ変数に1を足すと、ポインタのデータ型のサイズ分だけ先を参照するようになることを覚えておきましょう。

Section 23

コマンドライン引数

覚えておきたいキーワード
- コマンドライン引数
- main()関数

プログラムを実行する場合にはコマンドプロンプトで実行可能ファイルの名前を入力しますが、それに続けて文字列を入力すると、プログラム内でその文字列を利用することができます。このセクションではプログラムへの文字列の渡し方や受け取り方を解説します。

1. コマンドライン引数とは…

「**コマンドライン引数**」とは、ユーザーが任意でプログラムに与えることができる文字列のことです。コマンドライン引数をプログラムに与えるには、コマンドプロンプト上で実行可能ファイルの名前を入力した後、スペースを空けて文字列を入力します。その文字列が、そのままコマンドライン引数となります。

また、コマンドライン引数はスペースで区切っていくつでも指定できます。

構文 コマンドライン引数の利用

```
プログラム名　コマンドライン引数1　コマンドライン引数2　……　コマンドライン引数n
```

また、プログラム側でもコマンドライン引数を受け取るためのコードが必要です。コマンドライン引数を受け取るためには、**main()**関数で引数を受け取るように次のように記述します。

構文 コマンドライン引数を利用するmain()関数

```
int main(int argc, char *argv[])
```

変数名はこの名前にする必要はありませんが、構文に示した変数名を使うのが一般的です。コマンドライン引数を指定してプログラムを実行すると、「**argc**」には「コマンドライン引数の数」が入り、「**argv**」には「コマンドライン引数で渡された文字列」を参照するために必要な情報が入ります。

引数**argv[]**は、「**char**型のポインタの配列」という意味です。少々ややこしいですが、**char**型のポインタを、文字列を格納した文字型配列の先頭アドレスと考えれば、「**char *argv[]**」を次のような文字列の二次元配列として扱うことができます。

図1　文字列の2次元配列の例

str[0]	'H'	'e'	'l'	'l'	'o'	'¥0'	
str[1]	't'	'e'	's'	't'	'¥0'		
str[2]	's'	'u'	'z'	'u'	'k'	'i'	'¥0'

配列**argv**は、コマンドライン引数と次のように対応します。

表1　配列argvとコマンドライン引数

配列要素	内　容
argv[0]	プログラムの実行可能ファイル名（パス名を含む）を指すポインタ
argv[1]	コマンドライン引数1を指すポインタ
argv[2]	コマンドライン引数2を指すポインタ
⋮	⋮

コマンドライン引数がどのように対応するかを確かめるため、次のようなプログラムを作成してみましょう。このプログラムでは、**main()**関数がコマンドライン引数として受け取った文字列を順番に表示します。

Sample0605.c　コマンドライン引数

```
01  #include <stdio.h>
02
03  int main(int argc, char *argv[])
04  {
05      int i;
06      for(i=0; i<argc; i++){
07          printf("%s¥n", argv[i]);
08      }
09      return 0;
10  }
```

各コマンドライン引数を指すポインタを使います。

このプログラムを、コマンドライン引数を指定せずに実行すると、次のような結果になります。

画面に表示されるのは、プログラムを実行するために入力した実行可能ファイル名です（パス名を含みます）。つまり、**argv[0]**が指す文字列のみが表示されたことになります。

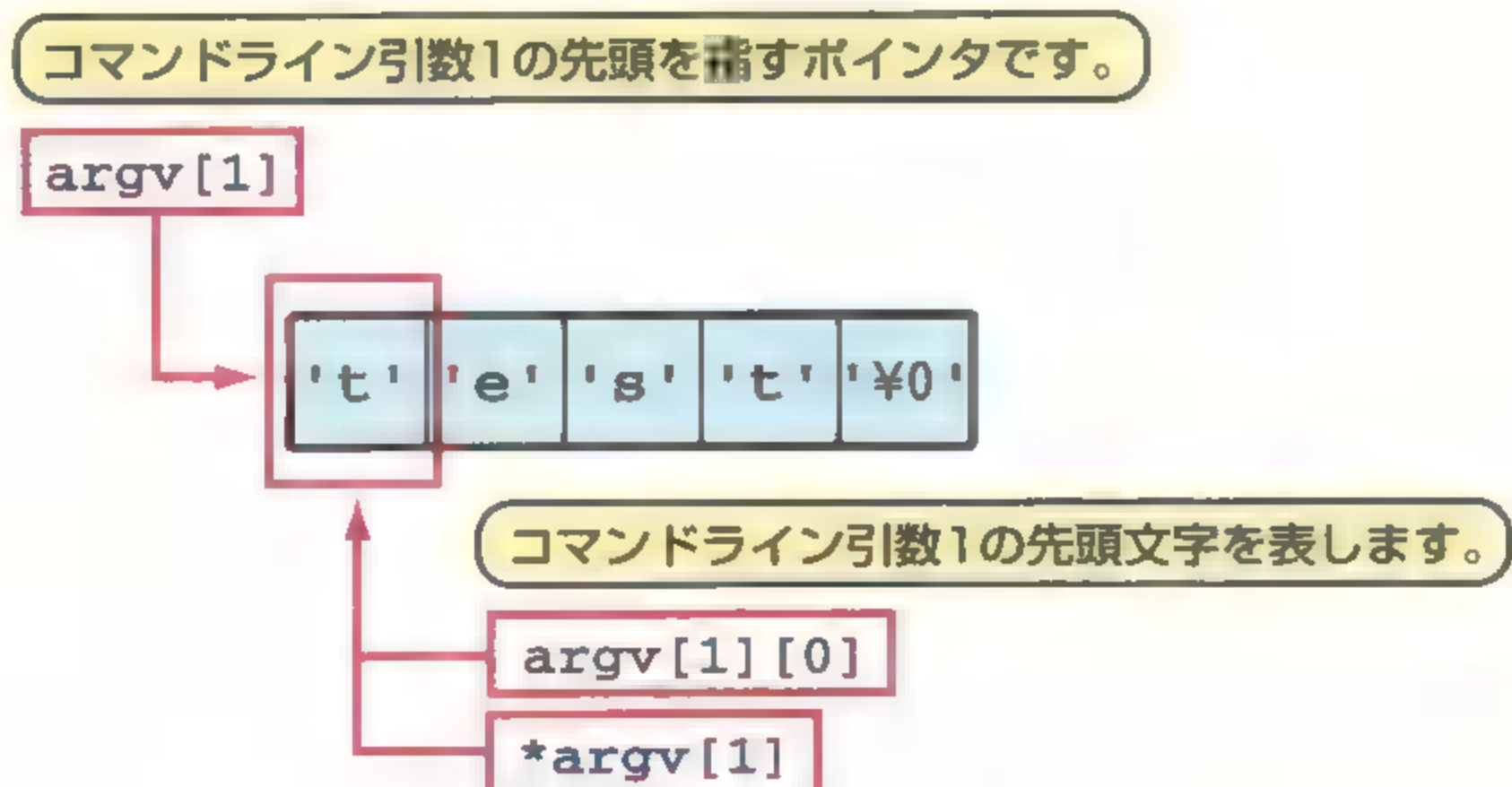
では、次にコマンドライン引数を指定して実行してみましょう。ここではコマンドライン引数1として「test」、コマンドライン引数2として「suzuki」と入力してみます。

```
C:¥work>sample0605 test suzuki
C:¥work¥sample0605.exe
test
suzuki

C:¥work>
```

今度は**argv[0]**が指す実行可能ファイル名に加え、**argv[1]**と**argv[2]**が指す「test」と「suzuki」という文字列がそれぞれ表示されます。

プログラムによって必要な引数の数が異なるため、コマンドライン引数はとりあえず、引数の数と引数の文字列を指すポインタの配列を受け取ります。そして、その中からプログラムで利用するデータを探してアクセスします。

たとえば、コマンドライン引数1に指定した文字列の各要素にアクセスしたい場合には、次のように**argv**を利用します。

図2　コマンドライン引数の文字列の取得方法

たとえば、これを**printf()**関数で次のように利用します。

```
printf("%s¥n", argv[1]);
printf("%c¥n", argv[1][0]);
printf("%c¥n", *argv[1]);
```

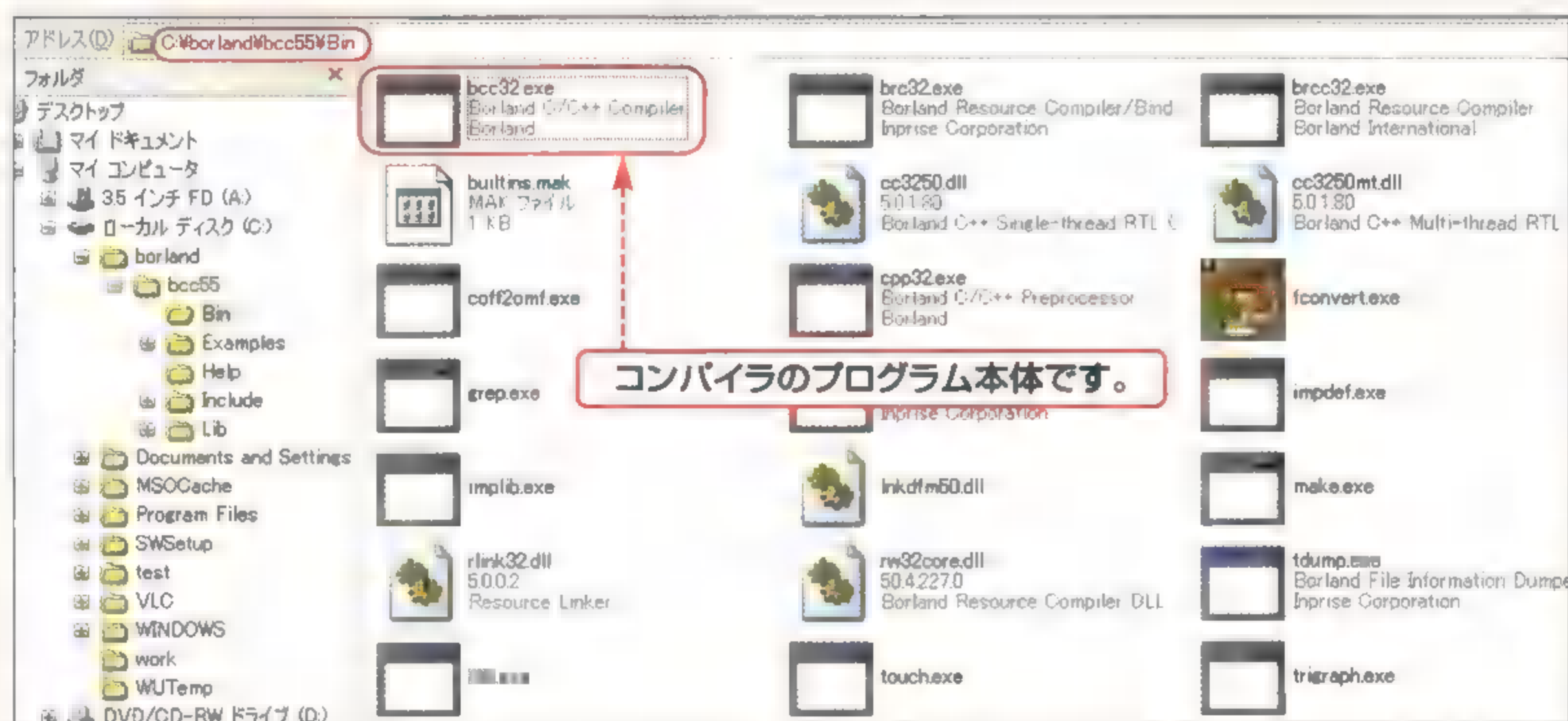
文字列「test」が表示されます。

文字「t」が表示されます。

Column コンパイラとコマンドライン引数

コンパイルを行うときに、コマンドプロンプト上で「bcc32 ソースファイル名」と入力していますが、この「ソースファイル名」の部分が、実はコマンドライン引数です。

「bcc32」というのはコンパイラのプログラム本体で、正確には「bcc32.exe」というファイル名です。このファイルはC:¥borland¥bcc55¥Binフォルダにあります。

コンパイラのプログラム本体です。

普段コンパイルを行うときは、次のように入力します。

```
bcc32 sample0101.c
```

ここでは、「sample0101.c」が「コマンドライン引数1」となります。スペースで区切ってコマンドライン引数2や3などを指定することもできます。すると、すべてのソースファイルをコンパイルし、1つのプログラムファイルを作成することができます（複数のソースファイルのコンパイルについてはSec.34参照）。

このようにコマンドライン引数を利用すると、プログラムの実行時に利用するファイルをユーザーが指定できるなど、自由度の高いプログラムが作成できます。

Section 24

関数のポインタ

覚えておきたいキーワード
- 関数ポインタ
- 関数ポインタの配列
- コールバック関数

変数がメモリ領域に確保されるように、関数もメモリ領域に確保されます。つまり、関数として確保されたメモリ領域のアドレスを、ポインタに代入することも可能です。このセクションでは、関数をポインタに格納して利用する方法などを解説します。

1. 関数のポインタの利用

■ 関数のポインタとは…

「関数のポインタ」とは、文字どおり「関数の処理が保存されているメモリ領域のアドレス」を格納するポインタです。これを「**関数ポインタ**」といいます。

変数と同様に、関数も一連の処理の流れがメモリ上に保存されます。この関数のアドレスを利用して、関数を呼び出すことが可能です。

ただし、関数ポインタは、関数を利用する上でもっとも複雑なしくみを持つ機能です。関数ポインタを利用する必要がある機会は、入門レベルのプログラミングではほとんどありません。本セクションの解説内容をすぐに理解する必要はないといってよいでしょう。「C言語では、こんなことも可能」という認識を持つだけでかまいません。

■ 関数ポインタの宣言

関数ポインタを利用するには、変数のポインタと同様に「関数型のポインタ」を宣言する必要があります。関数型のポインタを宣言するには、次のように記述します。

構文　関数ポインタ

```
戻り値の型 (*ポインタ名)(引数1, 引数2, …… ) = 関数名;
```

非常に複雑でややこしく見えますが、ほとんど関数宣言の形と変わりません。これを理解するには具体例を見るのが一番の近道です。関数ポインタを利用したプログラムは、次のようになります。

Sample0606.c 関数ポインタ

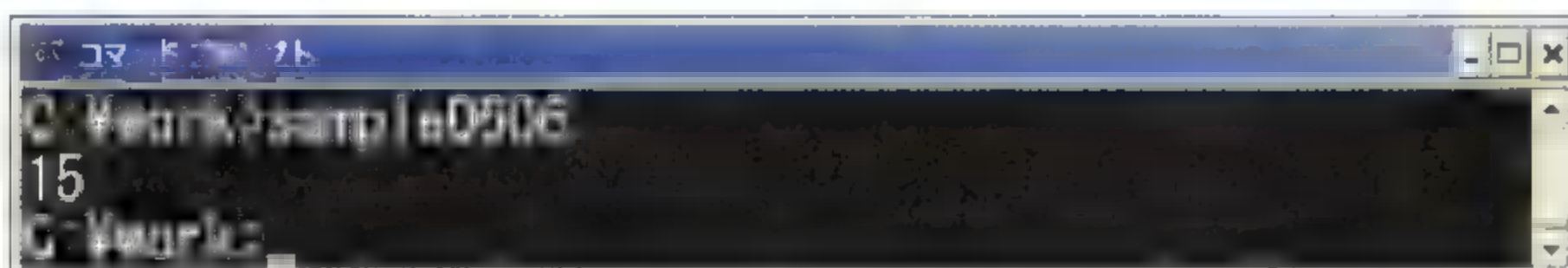
```
01  #include <stdio.h>
02
03  int plus(int a, int b)
04  {
05      return (a + b);
06  }
07
08  int main()
09  {
10      int num;
11      /* 関数のポインタ */
12      int (*pfunc)(int, int) = plus;
13      num = pfunc(10, 5);
14      printf("%d", num);
15
16      return 0;
17  }
```

この関数のアドレスをポインタに格納します。

「pfunc」がポインタ名です。

int型の引数が2つで、戻り値がint型の関数のポインタという意味です。

plus( )関数の関数名だけを記述すると、関数のアドレスを表現できます。

関数ポインタから関数を実行するには、「ポインタ名」+「( )」と記述します。

なお、関数ポインタの宣言と関数アドレスの代入は、分けて記述することもできます。

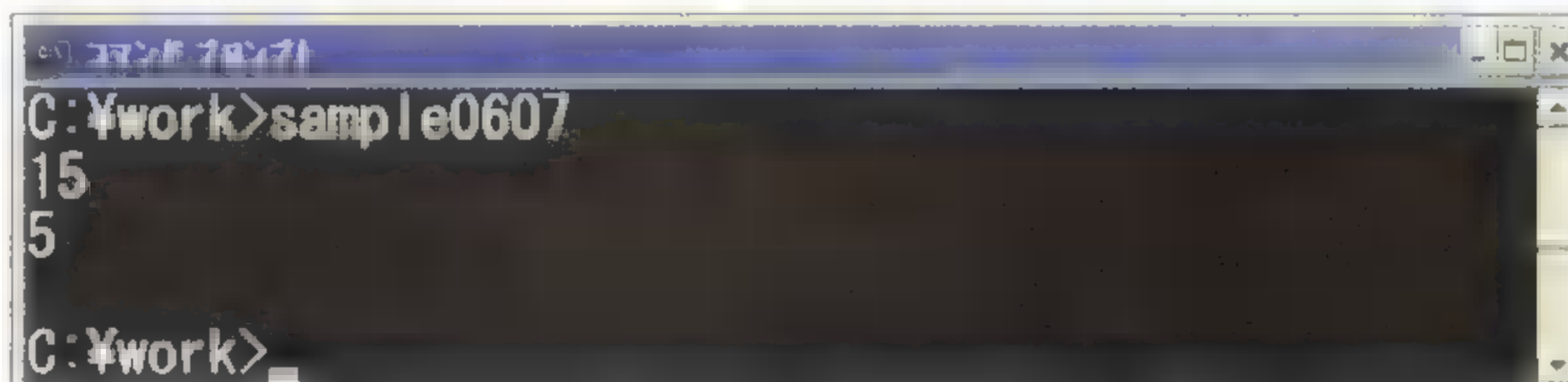
```
int (*pfunc)(int, int);
pfunc = plus;
```

■ 関数ポインタの配列

関数ポインタは、配列にして宣言することもできます。関数ポインタの配列は、次のように記述します。

構文 関数ポインタの配列

```
戻り値の型 (*ポインタ名[配列要素数])(引数1, 引数2, …… );
```

関数ポインタも通常のポインタと同じように配列の要素数を省略したり、宣言と同時に関数のアドレスを代入したりすることができます。

関数ポインタの配列を利用したプログラムを次に示します。

Sample0607 関数ポインタの配列

```
01  #include <stdio.h>
02  int plus(int a, int b)
03  {
04      return (a + b);
05  }
06
07  int minus(int a, int b)
08  {
09      return (a - b);
10  }
11
12  int main()
13  {
14      int i;
15      int num;
16      /* 関数のポインタ */
17      int (*pfunc[2])(int, int) = { plus, minus };
18      for(i=0; i<2; i++) {
19          num = pfunc[i](10, 5);
20          printf("%d¥n", num);
21      }
22      return 0;
23  }
```

引数と戻り値の型が同じ関数であれば、アドレスを代入することができます。

関数ポインタの配列から関数を実行するには、このように記述します。

```
C:¥work>sample0607
15
5

C:¥work>
```

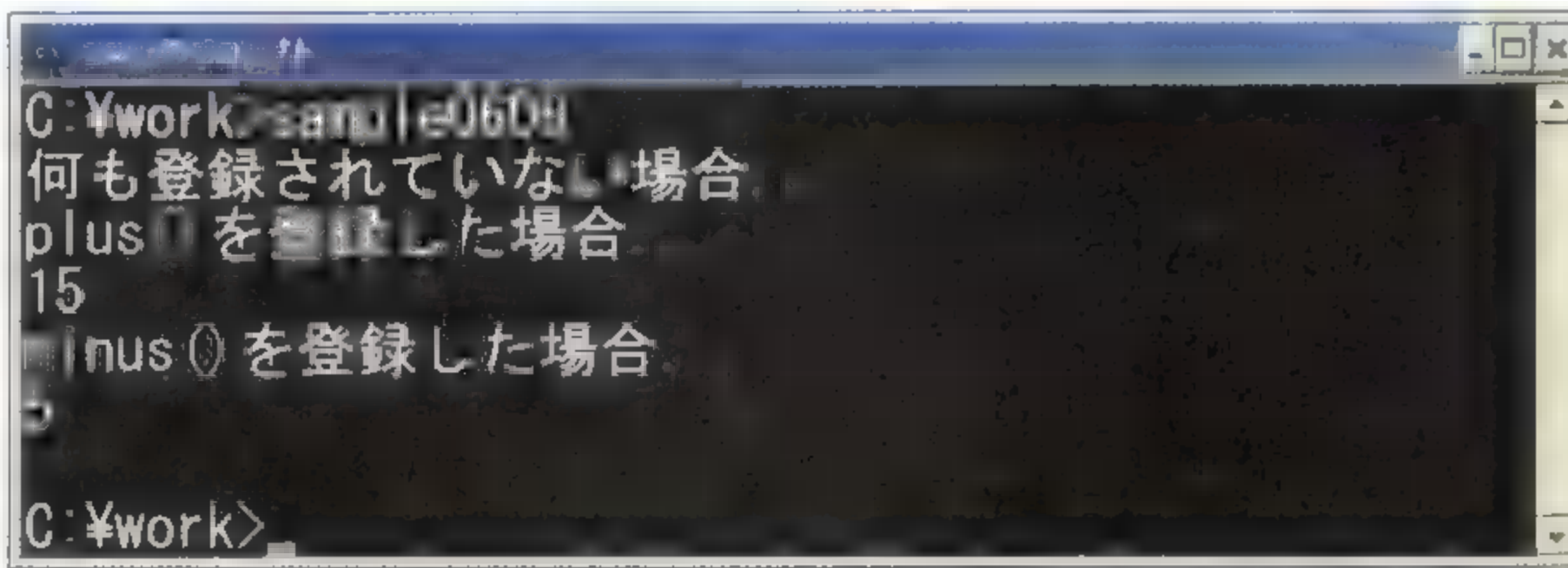
関数ポインタの配列を利用すると、同じ記述を使って関数をまとめて処理させやすくなります。また、呼び出す関数の追加や削除なども可能になります。

■ コールバック関数

「**コールバック関数**」とは、「特定の処理を実行した場合に呼び出してほしい関数」のことをいいます。外部から呼び戻されるというイメージがコールバック関数の名前の由来です。コールバック関数を利用するには、関数を呼び出したい処理において、関数ポインタを利用して登録しておいた関数を実行します。

コールバック関数を使用したプログラムは、次のようになります。

Sample0608.c コールバック関数

```
01  #include <stdio.h>
02
03  /* 関数のポインタ(グローバル変数) */
04  int (*pfunc)(int, int) = 0;
05
06  int plus(int a, int b)
07  {
08      return (a + b);
09  }
10
11  int minus(int a, int b)
12  {
13      return (a - b);
14  }
15
16  /* もし関数が登録されていたら実行する関数 */
17  void dofunc(void)
18  {
19      if( pfunc != 0 ){
20          int num;
21          num = pfunc(10, 5);
22          printf("%d¥n", num);
23      }
24  }
25
26  int main()
27  {
28      printf("何も登録されていない場合...¥n");
29      dofunc();
```

関数のアドレスが登録されているかどうかを判断するため。値0で初期化しておきます。

pfuncが0の場合は、関数が登録されていないので、処理ができません。

```
30
31      /* pfuncにplus()関数を登録する */
32      pfunc = plus;
33      printf("plus()を登録した場合...¥n");
34      dofunc();
35
36      /* pfuncにminus()関数を登録する */
37      pfunc = minus;
38      printf("minus()を登録した場合...¥n");
39      dofunc();
40
41      return 0;
42  }
```

関数ポインタとして何も登録していないと、19行目の**if**文で「**pfunc**が0」になるため、処理が実行されずに関数を抜けます。ポインタに代入する「値0」のことを、特別に「**NULL**」や「**NULL**ポインタ」などと呼びます。また、ポインタが0かどうかをチェックすることを「**NULL**チェック」といいます（次頁コラム参照）。

次に、**pfunc**に**plus()**関数のアドレスを代入すると**pfunc**が0ではなくなるため、20行目からの処理が実行され、**pfunc**に代入された**plus()**関数が実行されます。画面には、**plus()**関数の戻り値である「15」が表示されます。

最後に、**pfunc**に**minus()**関数のアドレスを代入した場合、21行目では今度は**minus()**関数が実行され、画面には**minus()**関数の戻り値である「5」が表示されます。

Column NULLポインタ

関数ポインタに限らず、一般的にポインタはNULLで初期化します。これは、そのポインタの指すアドレスが有効な領域かどうかをわかりやすくするためです。
ポインタをNULLで初期化していることをより明確に示すために、実はプログラム中で「NULL」という定数を使うことができます。ただし、NULLを利用するにはstdio.hなどをinclude文(Sec.34参照)で取り込む必要があります。
NULLは値0と同じ意味です。NULLを利用するには、次のように記述します。

```
int *p = NULL;
```

ポインタpにNULL(値0)を代入しています。

たとえば、関数ポインタのNULLチェックは次のように記述していましたが、

```
if( pfunc != 0 ){
        ⋮
```

値0の代わりにNULLを利用して、次のように記述できます。

```
if( pfunc != NULL )
        ⋮
```

値0の代わりにNULLを利用しています。

NULLを利用すると「ここではポインタがNULLかどうかを確認している」ということが明確になり、数値で0と記述するよりもソースコードの意味がわかりやすくなります。
ただし、NULLを整数型変数などに代入するのはやめましょう。NULLの用途はあくまで「初期化したポインタ」という意味を示すことで、変数に対して利用すべきではありません。また、整数型の変数なのかポインタなのかがわからなくなり、非常に読みにくいコードになります。

```
int n = NULL;
       ⋮
if( n == NULL ){
       ⋮
```

nは整数なのかポインタなのかわからなくなります。

まとめ　第6章：ポインタ

この章では、アドレスという概念と、それを格納するポインタについて学習しました。また、配列とポインタの違いについても学びました。コマンドプロンプトからプログラムに文字列を渡すコマンドライン引数についても触れました。

第6章で学習したこと

- アドレスとは、変数が確保したメモリ領域を指し示すものである。
- アドレス情報はポインタに格納できる。
- 配列は連続したメモリ領域に配置される。
- 配列名だけを記述した場合は、配列の先頭要素のアドレスを表す。
- ポインタも添え字によるアクセスに利用できる。
- ポインタを関数の引数に利用すると、関数の呼び出し元の変数の値を関数内で書き換えることができる。
- 関数にもアドレスがあり、関数ポインタに格納することができる。
- 関数ポインタの配列を利用して、関数を呼び出すことができる。

ステップアップ！

ポインタは昔から「C言語入門者の壁」といわれてきました。確かにこれまで学習してきたことよりコンピュータの世界に一歩踏み込んだ内容であるとは思います。しかし、難しく考えることなく、焦らずにじっくりと学習することが大切です。

本書を最後まで読み終え、C言語のプログラミングをある程度理解したとしても、ポインタやアドレスの考え方にはピンとこないかもしれません。ポインタについては、自分でプログラムを作成して扱い方を習熟させていくのが一番の近道です。ポインタを扱ったプログラムをできるだけ作成してみましょう。

問1 ポインタの利用

intの型の変数に、ポインタを利用して値を設定してください。

答1

解答例は、次のようになります。

```
int main()
{
    int n;
    int *p;
    p = &n;
    *p = 10;

    return 0;
}
```

問2 ポインタの引数への利用

引数としてint型のポインタを受け取り、関数の呼び出し元の変数を「0」に設定する関数を作成してください。

答2

ポインタが指し示す変数を利用するには、引数としてポインタを受け取ります。関数の定義では、データ型と、先頭に「*（アスタリスク）」を付けたポインタ名を記述します。

```
void zero(int *p)
{
    *p = 0;
}
```

問3　ポインタへの添え字の利用

文字列を**char**型のポインタとして受け取り、そのポインタから10文字か、NULL文字（'¥0'）が現れるまで画面に1文字ずつ表示する関数を作成してください。画面に1文字表示するたびに改行を入れてください。

答3

関数において引数として文字列を指すポインタを受け取り、添え字を利用して文字列の各要素にアクセスします。このとき、ループ処理を使って最後のNULL文字かどうかのチェックを行います。

```
void disp(char* str)
{
    int i;

    for(i=0; i<10; i++) {
        /* NULL文字だったらループを抜ける */
        if(str[i] == '¥0') {
            break;
        }
        printf("%c¥n", str[i]);
    }
}
```

第7章

Visual Learning Introduction of C

構造体

Section 25

構造体とは

覚えておきたいキーワード

- 構造体とメンバ
- ドット演算子
- 構造体の配列

構造体を利用すると、いくつかのデータ型をまとめて扱うことができます。扱うデータの種類が増えてくると、どの変数が何を意味するものなのか覚えきれなくなります。構造体を利用すると、必要なデータを1つにまとめて扱えるため、便利です。

1. 構造体の利用

構造体とは…

「**構造体**」とは、さまざまなデータ型を組み合わせて作る「新たなデータ型」のようなものです。たとえば、売店の商品リストを作ろうと思った場合を考えてみましょう。商品リストには「商品名」や「価格」などのデータが必要です。これらの必要な要素を1つにまとめて「商品リスト型のデータ」として便利に扱えるようにしたものが構造体です。

図1 構造体

構造体の定義

構造体を利用するには、まず構造体の型の定義が必要です。構造体の定義では、構造体の中に必要となるデータの変数を宣言します。この変数は、「**メンバ**」や「**メンバ変数**」と呼ばれます。

構造体を定義するには、次のように記述します。

構文 構造体の定義

```
struct 構造体の型名 {
    構造体のメンバのデータ型    メンバ名;
    ⋮
};
```

7 構造体

たとえば、先ほどの例のように売店の商品リストを構造体として作成してみましょう。商品リストには、「商品名」と「価格」のデータが必要になるので、次のような定義になります。

```
struct Goods {
    char name[30];      /* 商品名は30文字まで */
    int price;          /* 価格 */
};
```

「Goods」という名前の型として定義します。

■ 構造体変数の宣言

定義した構造体を利用するには、通常のデータ型と同じように変数宣言が必要です。構造体の変数宣言を行うには、次のように記述します。

構文　構造体変数の宣言

```
struct 構造体型名 変数名;
```

たとえば、**Goods**型の構造体変数を宣言するには、次のように記述します。

```
struct Goods g;
```

構造体変数は、通常の変数と同様に、宣言時に初期値を代入することも可能です。構造体変数を初期化するには、配列を初期化するときのように**{ }**の中に値を記述します。**{ }**には、構造体のメンバ変数に対応する値を「**,**（カンマ）」で区切って指定します。

たとえば、**Goods**型の構造体変数**g**を宣言し、商品名と価格にそれぞれ「鉛筆」と「30」という初期値を設定するには次のように記述します。

```
struct Goods g = {"鉛筆", 30};
```

メンバ変数name[30]に代入する文字列を指定します。

メンバ変数priceに代入する整数を指定します。

また、通常のデータ型と同様に、構造体型の配列を宣言することもできます。次のように記述すると、**Goods**型の構造体を10個含む配列が作成されます。

```
struct Goods gs[10];
```

■ メンバ変数の利用

構造体には「変数名」の他に「メンバ変数」という要素があり、構造体を利用する際には構造体に含まれる個々のメンバ変数にもアクセスしなければなりません。それぞれのメンバ変数にアクセスするためには「.（ピリオド）」を利用します。「.」は「**ドット演算子**」といい、次のように記述してメンバ変数にアクセスします。

構文　メンバ変数へのアクセス

```
変数名.メンバ変数名;
```

たとえば、**Goods**型の変数**g**を宣言し、**Goods**型の構造体に含まれるメンバ変数**price**に値「10」を代入する処理は次のようになります。

```
struct Goods g;
g.price = 10;
```

また、構造体型の配列において、配列要素の構造体のメンバ変数にアクセスするには、次のように記述します。

構文　配列要素の構造体のメンバ変数へのアクセス

```
変数名[配列要素番号].メンバ変数名;
```

構造体の配列も通常のデータ型の配列と同様に、宣言時に初期化することができます。1つの構造体を初期化する際には、メンバ変数の初期値を{}でくくって指定することを思い出してください。また、配列の初期化でも、配列に含まれる個々の要素の初期値を{}でくくって指定します。従って、構造体の配列を初期化するためには、次のように{}を入れ子にして初期値を記述します。

```
struct Goods gs[3] = {
    {"鉛筆", 30},
    {"消しゴム", 50},
    {"ノート", 100}
};
```

最初の要素のメンバ変数name[30]とpriceに初期値が代入されます。

各要素の構造体ごとに、初期値を指定します。

2. 構造体の活用

実際にGoods型の構造体の配列で商品リストを作成するプログラムの例を次に示します。

Sample0701.c　構造体の定義

```
01 #include <stdio.h>
02
03 /* 構造体の定義 */
04 struct Goods {
05     char name[30];     /* 商品名は30文字まで */
06     int price; /* 価格 */
07 };
08
09 int main()
10 {
11     int i;
12     /* Goods型の配列の宣言 */
13     struct Goods gs[3] = {
14         {"鉛筆", 30},
15         {"消しゴム", 50},
16         {"ノート", 100}
17     };
18
19     /* 画面に商品リストを表示する */
20     for(i=0; i<3; i++){
21         printf("商品名:%s¥t価格:%3d円¥n",
22                gs[i].name, gs[i].price);
23     }
24
25     return 0;
26 }
```

「¥t」はタブを表すエスケープシーケンスです（P.39参照）。

「%3d」は数値を右詰めにして表示する変換指定子です（P.37参照）。

```
コマンドプロンプト
C:¥work>sample0701
商品名:鉛筆      価格: 30円
商品名:消しゴム  価格: 50円
商品名:ノート    価格:100円

C:¥work>
```

Section 26

覚えておきたいキーワード
- 構造体のポインタ
- アロー演算子
- 関数の引数と構造体

構造体とポインタ

変数と同じように、構造体もポインタを使ってアクセスすることができます。関数の引数として構造体の変数を使う場合と、構造体のポインタを利用する場合を比べると、ポインタを利用する方がデータの無駄なコピーが行われないため、処理が速くなります。

1. メンバへのアクセス

ポインタの宣言

構造体も、他のデータ型と同様にポインタを利用することができます。構造体型のポインタを宣言するには次のように記述します。

構文 構造体のポインタ

```
struct 構造体型名 *ポインタ名;
```

また、構造体変数のアドレスを表現するには、変数と同様に、

構文 構造体変数のアドレス

```
&構造体変数名
```

のように記述します。たとえば、Goods型の構造体gのアドレスを格納するポインタを宣言する場合、次のようになります。

```
struct Goods *pg = &g;
```

ポインタを使ったメンバへのアクセス

構造体のメンバにアクセスするにはドット演算子を利用します。しかし、構造体型のポインタを使ってメンバにアクセスする場合はドット演算子を利用できません。代わりに「**->**（ハイフンと不等号）」を利用します。「**->**」は「**アロー演算子**」といいます。

アロー演算子を利用してメンバにアクセスするには、次のように記述します。

構文 メンバ変数へのアクセス(ポインタ)

```
ポインタ名->メンバ名;
```

構造体とドット演算子の関係が、そのまま構造体のポインタとアロー演算子に置き換わったと考えるとよいでしょう。これを利用して構造体型のポインタによりGoods型の構造体のメンバにアクセスするには、次のように記述します。

```
struct Goods *pg = &g;
pg->price = 30;
```

アロー演算子でメンバ変数priceにアクセスします。

2. 関数の引数としての構造体

関数の引数に構造体を利用する

構造体はメンバ変数を持つため、通常のデータ型に比べるとサイズが大きくなります。関数の引数は「コピー渡し(P.109参照)」であるため、もし構造体をそのまま関数の引数に利用すると、関数を呼び出すたびにすべてのメンバ変数の値がコピーされてしまいます。

コンピュータはそれくらいのコピーは目にもとまらぬ速さで処理してくれますが、しかし無駄なコピーを行ってたとえ微々たるものでも処理時間を増やしてしまうことは避けるべきです。規模の大きなプログラムになると、この微々たる違いが積み重なり、プログラムの効率を大幅に落とす大きな原因になることがよくあるからです。

図1 構造体の引数

これに対して、構造体のポインタを引数として利用する場合、コピーする情報は構造体を指し示すアドレス情報の4バイトだけですみます。つまり無駄なコピーが発生しないのです。そのため、関数に構造体を渡したい場合は、一般に引数として構造体のポインタを利用します。

図2 構造体のポインタの引数

■ 関数の引数に構造体のポインタを利用する

引数に構造体のポインタを利用する関数は、次のように記述します。

```
void foo(struct Goods *pg)
{
    ⋮
```

関数では、受け取ったポインタとアロー演算子（->）を利用して構造体のメンバ変数にアクセスすることができます。通常のデータ型のポインタを引数として渡す場合と同じように、ポインタとアロー演算子（->）を使ってメンバ変数の値を変更すると、呼び出し元でも変更が反映されます。

■ 関数の戻り値とポインタ

関数に構造体のデータを渡したい場合、ポインタを利用すると前項で説明しました。同様に、関数の戻り値としてポインタを返すこともできます。

たとえば、Goods型の構造体の配列を引数として受け取り、配列の中でもっとも高い商品を含む要素を探し、その要素のアドレスを戻り値として返す関数を定義します。

Sample0702.c 戻り値のポインタ

```
01  #include <stdio.h>
02
03  struct Goods {
04      char name[30];     /* 商品名は30文字まで */
05      int price;
06  };
07
08  struct Goods* foo(struct Goods gs[])
09  {
10      int max = 0;
11      int idx = 0;
12      int i;
13      /* もっとも値段の高い商品を探す */
14      for(i=0; i<5; i++) {
15          if( max < gs[i].price ) {
16              max = gs[i].price;
17              /* もっとも値段の高い商品の
18                 添え字を記憶しておく */
19              idx = i;
20          }
21      }
22      /* もっとも値段の高い商品のアドレスを返す */
23      return &gs[idx];
24  }
25
26  int main()
27  {
28      struct Goods *pg;
29      struct Goods gs[5] = {
30          {"鉛筆", 30},
31          {"消しゴム", 50},
32          {"ノート", 100},
33          {"ボールペン", 80},
34          {"クリップ", 20}
35      };
36      pg = foo(gs);
37      printf("もっとも値段の高い商品は¥n");
38      printf("%s(%d円)¥n", pg->name, pg->price);
```

配列gsの中から、priceの値のもっとも大きいものを探します。（15行目）

構造体の配列の要素のアドレスを戻り値として返します。（23行目）

構造体型のポインタを宣言します。（28行目）

関数を呼び出し、その戻り値をポインタに格納します。（36行目）

```
39      printf("です。");
40
41      return 0;
42  }
```

```
C:¥work>sample0702
もっとも値段の高い商品は
ノート（100円）
です。
C:¥work>_
```

この例では、受け取った構造体の配列の各要素についてメンバ変数**price**を調べ、もっとも値段の高い商品の情報を含む要素のアドレスを戻り値として返す関数を定義しています。

関数の呼び出し元では**Goods**型のポインタを宣言し、関数を呼び出してその戻り値を格納します。このポインタを使えば、もっとも値段の高い商品を含む構造体のメンバ変数に簡単にアクセスすることができます。

ただし、関数の戻り値として利用できないポインタもあることに注意が必要です。たとえば、関数内で作成したローカル変数に値を格納してそのアドレスを戻り値で返すことはできません。これは、一見正しい処理のように見えますが、このような処理は絶対に行ってはいけません。その理由は、スコープに関係があります。具体的な例を見てみましょう。

```
struct Goods {
    char name[30]; /* 商品名は30文字まで */
    int price;
};

struct Goods* foo(void)
    struct Goods g = {
        "ノート", 100
    };
    printf("%s, %d¥n", g.name, g.price);
    return &g;
}

int main()
```

スコープの範囲
1 ここで宣言した変数は、
2 ここでスコープから外れます。

```
{
    struct Goods *pg;
    pg = foo();
    printf("%s, %d¥n", pg->name, pg->price);
     ⋮
```

3 この時点で、受け取ったアドレスは無効になっているので、不正なメモリアクセスとなります。

関数の中で宣言した変数は、関数が終了すると同時にスコープから外れます。スコープから外れると、確保されていたメモリ領域が解放されて無効な変数となります。

foo()関数の戻り値として受け取ったポインタ**pg**は、すでに参照先のアドレスが無効な変数になっており、ポインタ**pg**にアクセスする処理は不正なメモリアクセスとなってしまいます。その結果、無効となったメモリ領域を参照してしまい、画面には壊れたデータが表示されています。こういったアクセスは重大なバグとなる可能性があるため、注意しましょう。

■ 関数の結果を構造体で返すには…

関数から構造体のデータを呼び出し元に返したい場合は、戻り値を返すための構造体のポインタを引数として受け取るようにします。関数の呼び出し元で確保しておいた構造体変数のアドレスを関数に渡し、関数の中でポインタを使ってこの構造体変数にデータを代入することで、呼び出し元に値を返すのが一般的です。

具体的には、次のように記述します。

```
void foo(struct Goods *pout)
{
    /* 引数のポインタを使って構造体にデータを代入する */
    pout->price = 100;
}
```

ポインタを使い、関数の呼び出し元で確保している変数へ値を代入します。

Section 27

共用体

覚えておきたいキーワード
- 共用体
- 構造体との違い
- 共用体のメモリ

C言語には、構造体とよく似たしくみを持つ共用体があります。共用体は、行う処理によって扱うデータの種類がまちまちである場合に、メモリ節約のために利用されます。共用体に宣言した複数のメンバは、同時に利用することはできないので注意が必要です。

1. 構造体との違い

■ 共用体の定義

「**共用体**」は、宣言の仕方も利用方法も構造体によく似ています。異なるのは「メモリの確保の仕方」です。まずは共用体の定義方法を確認しましょう。

構文　共用体の定義

```
union 共用体型名 {
     型   メンバ名;
      ⋮
};
```

構造体の「**struct**」が「**union**」に変わっただけです。共用体変数の宣言の記述は、次のようになります。

構文　共用体変数の宣言

```
union 共用体型名 変数名;
```

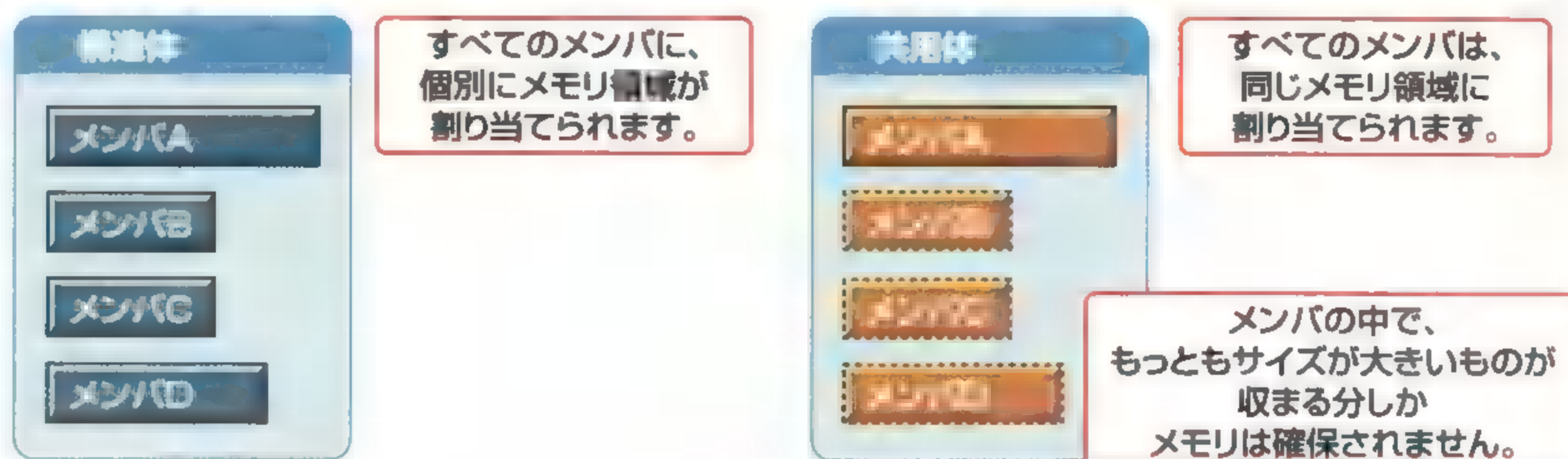
これも構造体とほぼ同じです。「共用体の配列」や「共用体のポインタ」なども構造体と同じ方法で利用することができます。ドット演算子やアロー演算子の使い方も構造体と同じです。

■ 共用体の確保するメモリ

ここまでは構造体と共用体の違いは、「**struct**」か「**union**」かという程度しかわかりません。構造体と共用体のもっとも大きな違いは、確保されるメモリの「量」です。

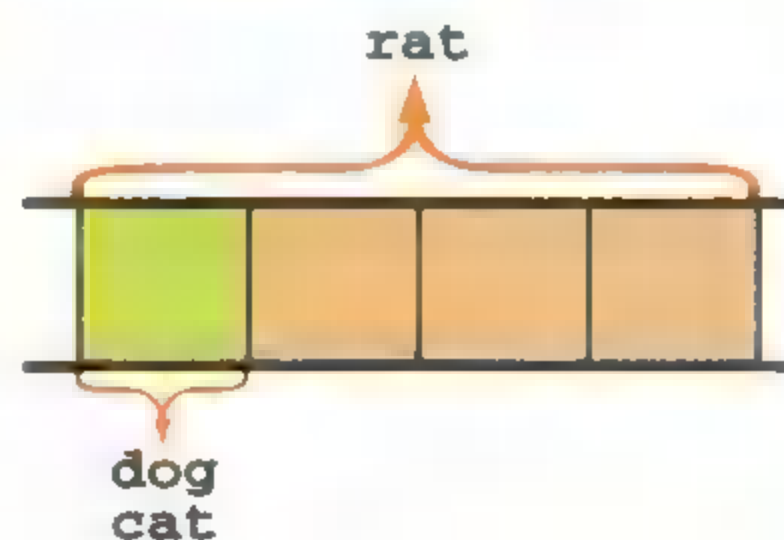
共用体では、メンバ変数ごとに個別のメモリは割り当てられません。すべてのメンバ変数が同

じアドレスに割り当てられます。確保するメモリ領域は、メンバの中でもっともサイズの大きなものに合わせて確保されます。

図1　構造体と共用体

たとえば、次のような共用体を宣言します。

```
union Animal {
    unsigned char dog;
    unsigned char cat;
    int rat;
}
```

この場合、**dog**と**cat**は**unsigned char**型なので1バイト、**rat**は**int**型なので4バイトとなり、共用体のサイズは「もっともサイズの大きいメンバ」に合わせるため、4バイトとなります。

図2　共用体具体例

そして、すべてのメンバ変数は「同じアドレス」に割り当てられます。たとえば変数**dog**に値10を代入すると、変数**cat**も変数**rat**も10になってしまいます。

実際にプログラムで確認してみましょう。

Sample [illegible] 共用体

```
01  #include <stdio.h>
02
03  union Animal {
04      unsigned char dog;
05      unsigned char cat;
06      int rat;
07  };
08
09  int main()
10  {
11      union Animal ani;
12      /* もっともサイズの大きいメンバで初期化しておく */
13      ani.rat = 0;
14
15      /* dogに10を代入する */
16      ani.dog = 10;
17      printf("dog = %d, cat = %d, rat = %d¥n",
18                  ani.dog, ani.cat, ani.rat);
19      /* dogに20を代入する */
20      ani.dog = 20;
21      printf("dog = %d, cat = %d, rat = %d¥n",
22                  ani.dog, ani.cat, ani.rat);
23      /* アドレスを表示する */
24      printf("&dog = %x, &cat = %x, &rat = %x¥n",
25                  &ani.dog, &ani.cat, &ani.rat);
26      return 0;
27  }
```

メンバ変数dogに10を代入し、すべてのメンバ変数を表示しています。

メンバ変数dogに20を代入し、すべてのメンバ変数を表示しています。

すべてのメンバ変数のアドレスを表示しています。

```
C:¥work>sample0703
dog = 10, cat = 10, rat = 10
dog = 20, cat = 20, rat = 20
&dog = 12ff88, &cat = 12ff88, &rat = 12ff88

C:¥work>
```

すべてのメンバの値が「10」になっています。

すべてのメンバの値が「20」になっています。

すべてのメンバが同じアドレスに割り当てられています。

16行目で**dog**に「10」を代入すると、共用体**ani**の中の**dog**、**cat**、**rat**のすべてが10になってしまいます。同様に20行目で**dog**に「20」を代入すると、すべてのメンバの値が20になります。これは、すべてのメンバが同じメモリ領域に割り当てられているためであり、24行目を実行するとすべての変数が同じアドレスに割り当てられているのがわかります。

■ 共用体の目的

これまでに見てきたとおり、共用体は複数のメンバを同時に利用することはできません。共用体の目的は「メモリの節約」にあります。

たとえば、「**A**という構造体と**B**という構造体について、それぞれ100個の要素を持つ配列が必要だ」という場合、構造体**A**や**B**のサイズが大きいとメモリ領域を大量に消費してしまいます。ここで「**A**と**B**の構造体は、両方を合わせて最大で100個しか使わない」という条件が加われば、共用体を利用することでメモリの節約が可能です。

すなわち、共用体のメンバに構造体**A**と構造体**B**を入れて、共用体を100個の配列にすればよいのです。具体的なプログラムにすると、次のようになります。

共用体を利用しない場合

```
struct A a[100];
struct B b[100];
```

構造体AとBをそれぞれ100個ずつ確保します。

共用体を利用する場合

```
union AandB {
    struct A a;
    struct B b;
}
:
union AandB ab[100];
```

構造体AとBをメンバに持つ共用体を100個確保すると、メモリの節約になります。

共用体は**A**と**B**のうちサイズの大きい構造体に合わせたサイズになるため、サイズの小さい方の構造体100個分のメモリを節約できます。

ただし、その共用体が構造体**A**として使われているのか構造体**B**として使われているのかは、プログラマが自分で管理しなければなりません。

Column リトルエンディアン

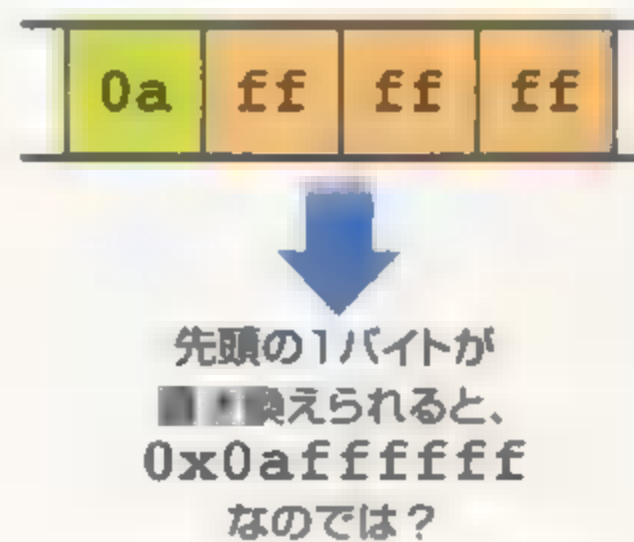
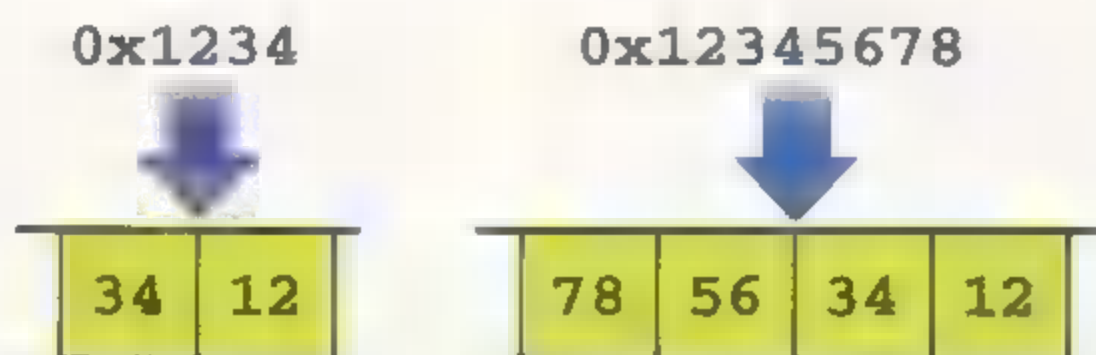
変数のアドレスには、ちょっとした不思議な性質があるので紹介しましょう。

P.164のプログラム例の13行目で共用体の初期化を行っていますが、これを次のように書き■えて実行してみてください。

```
/* もっともサイズの大きいメンバで初期化しておく */
ani.rat = 0xffffffff;
```

実行結果ではratの値が正しく表示されないように見えます。今度は17行目と、21行目にあるprintf()関数を書き換えて、変数の中身を16進数で表示してみます。

```
printf("dog = %x, cat = %x, rat = %x¥n", ani.dog, ani.cat, ani.rat);
```

今度は、dogとcatの値が「a」であるのに対して、ratの値が「ffffff0a」であるとわかります。これはdogやcatが1バイト、ratが4バイトであることが原因で起こる現象です。しかし、これらの変数はすべて同じアドレスに配置されているのに、なぜ4バイトの変数の一番右の1バイトが書き換えられるのでしょうか？（右図参照）

実は、Windowsの動作しているパソコンは2バイト以上の情報をメモリに保存するときに「下位バイトが左、上位バイトが右」になるように配置しています。メモリ上の配置方法は動作するパソコンやOSにより決まっています。

「下位バイトを左に、上位バイトを右に」保存する方式を「リトルエンディアン」といいます。逆に「上位バイトを左に、下位バイトを右に」保存する方式を「ビッグエンディアン」といいます。Windowsの動作するパソコンはリトルエンディアン方式を利用していますので、プログラム例ではもっとも左の1バイトが書き換えられ、その結果rat変数は0xffffff0aになったというわけです。

Column 1行コメント

前述のプログラム例のように、プログラミングを行う際にはプログラム中に処理の内容を説明するコメントを記述します。他の人だけでなく、自分で後から読んだときにも、そのプログラムで何を行っているのか処理の流れがひと目でわかるように適切なコメントを記述する習慣を身につけましょう。
P.11で説明したように、C言語でコメントを記述する場合、正しくは「/*」と「*/」で囲む方法しかありませんでした。その後、C++で「1行コメント」という仕様が追加され、これが非常に使い勝手がよかったため、C言語のコンパイラも導入していきました(ただし、C言語の機能としては規格外ですが)。それを追認する形で、1999年に新しいC言語の規約(C99、P.24参照)を作成した際に、1行コメントが正式にC言語の規約に盛り込まれました。
しかし、古いコンパイラでは利用できない場合があるので本書ではコラムで紹介するにとどめておきます。もし利用しているコンパイラが1行コメントに対応している場合、以下を参考に利用してみてください。

構文 1行コメント文

```
// コメント文
```

「//」から行末までがコメントとして扱われます。これを利用したプログラムは次のようになります。

```
// もっともサイズの大きいメンバで初期化しておく
ani.rat = 0xffffffff;
```

この部分がコメントになります。

なお、C++やJavaでもこの形式の1行コメントを使用することができます。

構造体でよく利用する演算子

構造体の変数を宣言する場合、毎回structを付けるのは面倒です。typedef演算子を利用すれば、structを省略することができます。また、構造体の合計サイズを取得したい場合は、sizeof演算子を利用します。sizeof演算子は通常の変数やデータ型にも利用できます。

1. typedef演算子

■ データ型に別名を付けるための演算子

構造体を利用する場合、構造体名の前に必ず**struct**を付けなければならないため、変数宣言などにおいてどうしても文が長くなります。「**typedef**演算子」を利用すると、これを短くわかりやすい名前に置き換えることができます。**typedef**演算子を利用するには、次のように記述します。

構文 typedef演算子

```
typedef 置き換えたいデータ型 新しい型名;
```

構造体GoodsをGOODSという記述でも宣言できるようにするには、次のように記述します。

```
typedef struct Goods GOODS;
```

「GOODS」を「struct Goods」と同じ意味であると宣言します。

新しく定義したGOODSを利用する場合、次のように「**struct**」を付けないで変数を宣言することができます。

```
GOODS g;
```

また、**typedef**で作成した新しい型名のポインタを作成したり、関数の引数にしたりすることもできます。

```
typedef struct Goods GOODS;

void foo(GOODS *pg);
```

ポインタの宣言や関数の引数に利用することができます。

なお、**typedef**演算子は構造体だけでなく、**int**などの標準のデータ型に対しても利用できます。

```
typedef int BOOLEAN;
typedef unsigned char BYTE;
```

標準のデータ型の別名を定義する際にも利用できます。

■ 構造体の定義と同時に行うtypedef

構造体を定義するのと同時に、**typedef**によって新しい型名を与えることもできます。定義と同時に新しい型名を与えるには、次のように記述します。

構文 typedef演算子（構造体の定義）

```
typedef struct 構造体名 {
    ⋮
} 新しい型名;
```

これを利用して構造体GoodsをGOODS型として定義するコードは次のようになります。

```
typedef struct Goods {
    char name[30];/* 商品名は30文字まで */
    int price;
} GOODS;
```

2. sizeof演算子

■ メモリ容量をはかる演算子

「**sizeof**演算子」を利用すると、指定した変数やデータ型が何バイトのメモリを使用しているかを知ることができます。この演算子により、配列が使用しているメモリ容量の計算も可能です。

sizeof演算子を利用するには、次のように記述します。

構文 sizeof演算子

```
sizeof(変数);
sizeof(配列名);
sizeof(データ型名);
```

sizeof演算子を利用したプログラム例を次に示します。

Sample0705.c sizeof演算子

```
01 #include <stdio.h>
02 
03 typedef struct Block {
04     char str[8];
05     int num;
06 } BLOCK;
07 
08 int main()
09 {
10     char c;
11     int i;
12     int pi[5];
13     BLOCK b;
14 
15     printf("char型 = %d¥n", sizeof(char));
16     printf("変数c = %d¥n", sizeof(c));
17     printf("int型 = %d¥n", sizeof(int));
18     printf("変数i = %d¥n", sizeof(i));
19     printf("配列pi = %d¥n", sizeof(pi));
20     printf("BLOCK型 = %d¥n", sizeof(BLOCK));
21     printf("変数b = %d¥n", sizeof(b));
22 
23     return 0;
24 }
```

データ型や変数のサイズを取得できます。

構造体や配列のサイズも取得できます。

```
C:¥work>sample0705
char型 = 1
変数c = 1
int型 = 4
変数i = 4
配列pi = 20
BLOCK型 = 12
変数b = 12

C:¥work>
```

Column パディング

たとえば、右に示す構造体のサイズを**sizeof**演算子を利用して表示するプログラムを作成してみましょう。

```
typedef struct Block2 {
    char str[10];
    int num;
} BLOCK2;
```

この構造体は、左頁のプログラム例に登場した構造体とほとんど同じです。違いは、メンバの配列**str**の長さが10になっていることです。
では、この構造体は何バイトのメモリを使うでしょうか? 計算上は**char**型を10個含む配列の10バイトと**int**型の4バイトで合計14バイトとなるはずです。しかし、プログラムを実行してみると、14バイトではなく16バイトと表示されます。

```
C:¥work>sample0706
BLOCK2型 = 16

C:¥work>
```

これはプログラミングのミスではなくC言語コンパイラが「勝手に」多めにメモリを確保しているのです。このように実際には使われないけれど確保されたメモリのことを「パディング」と呼びます。パディングは、データや構造体を配列にしたときに「先頭アドレスの値が必ず4バイト単位になる」ように、構造体の容量を調節するために確保されます。
4バイトというのはCPUがメモリ上のデータにアクセスする単位です(コンピュータのアーキテクチャによってアクセス単位が異なる場合があります)。そのため、変数の領域をメモリ上に確保する際に、4バイト単位に揃えるとアクセス速度が速くなります。従って、コンパイラはパディングを入れて、構造体の容量が「4の倍数」のバイト数になるように調整します。

構造体のサイズが4の倍数のバイト数になるようにパディングが挿入されます。

ただし、コンパイラがパディングを挿入するかどうかは、コンパイラやコンパイル時の設定に依存します。構造体のサイズを利用するようなプログラミングでは、必ずコンパイラによるパディングの挿入があるかどうかを確認しましょう。

まとめ 第7章：構造体

この章では、さまざまなデータを1つにまとめて扱える構造体について学習しました。構造体を利用すると、たとえば「名前」と「年齢」などのデータをひとまとめにして管理できるので便利です。また、場合によっては構造体よりもメモリを節約できる共用体についても学びました。

第7章で学習したこと

- 関連するデータを1つにまとめるには、構造体を利用する。
- 構造体の中に宣言した変数のことを「メンバ変数」と呼ぶ。
- メンバ変数にアクセスするには、構造体変数名の後に「.（ピリオド）」を付け、続けてメンバ変数名を指定する。
- 構造体もポインタを利用することができる。
- 構造体のポインタからメンバ変数にアクセスするには、「.」の代わりに「->（ハイフンと不等号）」を利用する。
- 構造体と似たしくみの共用体がある。場合によっては共用体を利用することでメモリが節約できる。
- typedef演算子を利用すると、構造体の宣言時に「struct」を使わなくてもよい。
- sizeof演算子を利用すると、データ型や変数のサイズを取得することができる。

ステップアップ！

構造体は、プログラムの中で扱うデータが増えるに従って必要になります。構造体を使わず、たとえば配列をいくつも用意してデータを管理しようとすると、非常に面倒です。

構造体を利用すると、関数を定義する際にたくさんの引数を並べる必要がない、2つ以上の情報を関数の呼び出し元に簡単に戻すことができるなど、非常に便利です。ポインタや関数と比べると単純なしくみであるので、しっかり覚えておきましょう。

問1　構造体の宣言

int型と**double**型の変数を1つずつメンバとして持つ構造体を宣言してください。また、**typedef**演算子を用いて、構造体変数を宣言する際に**struct**を付けなくてすむようにしてください。その構造体の変数を**main()**関数で宣言し、**int**型のメンバに「10」、**double**型のメンバに「1.0」を代入し、メンバ変数の内容を画面に表示してください。

答1

解答例は、次のようになります。

```
01  #include <stdio.h>
02
03  typedef struct number {
04      int i;
05      double d;
06  } NUMBER;
07
08  int main()
09  {
10      NUMBER num;
11      num.i = 10;
12      num.d = 1.0;
13
14      printf("%d, %f", num.i, num.d);
15      return 0;
16  }
```

問2　構造体のポインタ

問1で作成した構造体のポインタを受け取り、**int**型のメンバと**double**型のメンバにどちらも「0」を代入して、画面にメンバ変数の値を表示する関数を作成してください。

答2

構造体のポインタからメンバ変数にアクセスするには「**->**(アロー演算子)」を利用します。

```
typedef struct number {
    int i;
    double d;
} NUMBER;

void zero(NUMBER *p)
{
    p->i = 0;
    p->d = 0.0;

    printf("%d, %f", p->i, p->d);
}
```

問3 共用体の宣言

メンバに**int**型と**char**型の変数を持つ共用体を定義してください。

答3

共用体は「**union**」を利用して定義します。

```
union test {
    int i;
    char c;
};
```

第 8 章

Visual Learning Introduction of C

標準入出力ライブラリ

Section 29

キー入力の受け取り

- 標準入出力ライブラリ
- scanf()関数
- gets()関数

プログラムがユーザーからのキーの入力を受け取るには「標準入出力ライブラリ」を利用します。動作しているプログラムがキー入力を受け取れるようになると、ユーザーが実行する処理を選べるようになるなど、プログラムの表現の幅が広がります。

1. 標準入出力ライブラリの利用

■ 標準入出力ライブラリとは…

C言語では、よく使う関数をライブラリという1つのファイルにまとめ、それぞれのプログラムでライブラリの関数を利用することができます。ANSI Cの規格では、画面入出力、文字列操作、数学関連の計算など、用途の広い関数があらかじめ定義されており、**標準ライブラリ**として提供されています。

中でも、画面入出力を行うためのライブラリを「**標準入出力ライブラリ**」と呼びます。これまでにたくさんのプログラム例を紹介してきましたが、その冒頭に「`#include <stdio.h>`」と記述していたことを思い出してください。`stdio.h`が標準入出力ライブラリ用の**ヘッダーファイル**であり、`printf()`などの関数の定義を含んでいます。

`#include`文とライブラリのヘッダーファイルの詳細については、Sec.34を参照してください。

2. scanf()関数

■ scanf()関数とは…

プログラムにおいて、ユーザーのキーボード入力を受け取るには「`scanf()`関数」を利用します。`scanf()`関数は、`printf()`関数と似た書式でユーザーからの入力を受け取れるため、便利です。`scanf()`関数を利用するには、次の書式に従います。

構文 scanf()関数

```
scanf("変換指定子", &変数);
```

scanf()関数を利用して整数の入力を受け取るプログラムの例を次に示します。

```
int i;
scanf("%d", &i);
```

変数のアドレスを引数として渡します。

文字列を受け取る場合は、次のように記述します。

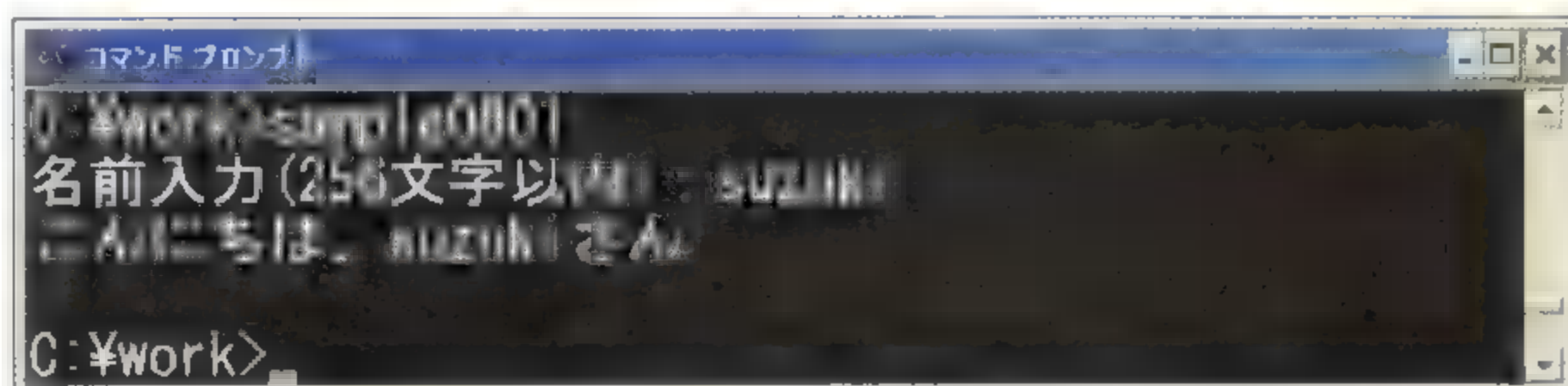
```
char str[256];
scanf("%s", str);
```

配列の先頭アドレスを引数として渡します。

なお、文字列を入力するための配列の長さは、入力される文字列よりも長くなければなりません。前述の例でも256文字以上の文字が入力されるとバッファオーバーフローになるため、**scanf()**関数を実行する前に**printf()**関数などで「256文字以上は入力できない」というメッセージを表示するなどの対処が必要です。

scanf()関数を利用してキーボードから入力を受け取るプログラムは、次のようになります。

Sample0801.c scanf()関数の利用

```
01 #include <stdio.h>
02 
03 int main()
04 {
05     /* 入力バッファ */
06     char str[256];
07 
08     printf("名前入力(256文字以内):");
09     /* キーボード入力を受け取る */
10     scanf("%s", str);
11 
12     printf("こんにちは、%sさん¥n", str);
13 
14     return 0;
15 }
```

■ スペース文字を扱えない

scanf()関数は、**スペース文字**を文字列として読み込めません。スペース文字が入力されると、文字列の読み込みを停止してしまいます。

スペース文字の入力を扱うためには、「**gets()**関数」を利用します。

3. gets()関数

gets()関数は**scanf()**関数と同様にキーボードからの入力を受け取る関数です。**scanf()**関数と違う点は、文字列の入力しか受け取れないことと、スペース文字を読み込めるということです。

gets()関数を利用するは、次のように記述します。

構文 **gets()関数**

```
gets(char型の配列名);
```

P.177のプログラムを**gets()**関数を利用して書き直すと次のようなプログラムになります。**gets()**関数を利用するとスペース文字も読み込めることに注目しましょう。

Sample0802.c gets()関数の利用

```
01  #include <stdio.h>
02
03  int main()
04  {
05      /* 入力バッファ */
06      char str[256];
07
08      printf("名前入力(256文字以内):");
09      /* キーボード入力を受け取る */
10      gets(str);
11
12      printf("こんにちは、%sさん¥n", str);
13
14      return 0;
15  }
```

```
C:¥work>sample0802
名前入力(256文字以内):suzuki kenichi
こんにちは、suzuki kenichiさん

C:¥work>
```

なお、**scanf()**関数も**gets()**関数も、日本語の入力を受け取ることもできます。

コマンドプロンプト上で日本語を入力するには、Alt+半角/全角を押します。この状態では、メモ帳などで日本語入力を行うのと同様に、漢字変換なども行うことができます。もう一度押すと、英数字入力に戻ります。

Alt+半角/全角キーを押すと
日本語入力が
行える状態になります。

画面への出力

画面への出力でも標準入出力ライブラリを利用します。今まで画面表示を行う際に利用してきたprintf()関数も、標準入出力ライブラリのひとつです。画面出力を行う関数も、入力を行う関数と同様にいくつか種類があり、用途に応じて適切なものを選択します。

- printf()関数
- puts()関数
- sprintf()関数

1. printf()関数

今までのプログラム例で何度も利用したので細かい説明は割愛します。**printf()**関数は、**変換指定子**を利用して整数や文字列などを適切な形式で画面に出力することができます。ただし、改行は自動的には行われません。

```
printf("整数:%d", 10);
printf("文字列:%s", "こんにちは");
```

改行は行われません。

2. puts()関数

puts()関数は、文字列を画面に出力する関数です。整数などは利用できません。**puts()**関数で文字列を出力すると、出力した文字列の最後に自動的に改行が付けられます。

構文 puts()関数

```
puts(char型の配列名);
```

puts()関数を利用したプログラムは、次のようになります。

```
char str[] = "おはよう";
puts(str);
puts("こんにちは");
```

文字列を格納したchar型の配列名を指定します。

このように文字列を直接指定することもできます。

Column sprintf()関数

画面出力を行う関数ではありませんが、**printf()**関数とよく似た関数として「**sprintf()**関数」を紹介します。

sprintf()関数は、画面出力は行えません。代わりに、「文字列への出力」を行うことができます。出力方法は**printf()**関数とほぼ同じなので、文字列を加工するのに便利です。**sprintf()**関数を利用するには、次の書式に従います。

構文 sprintf()関数

```
sprintf(出力先配列名, "……変換指定子……", 変数, 変数, ……);
```

sprintf()関数を利用するプログラムは、次のようになります。

Sample0805.c sprintf()関数の利用

```
01  #include <stdio.h>
02
03  int main()
04  {
05      char str[256];
06      char name[] = "ノート";
07      int price = 100;
08
09      sprintf(str, "商品:%s 価格:%d円", name, price);
10      puts(str);
11
12      return 0;
13  }
```

9行目の“”で囲まれた部分が、配列**str**に文字列として代入されます。宣言時以外で配列に文字列を代入することはできないので、**sprintf()**関数は文字列の加工には、たいへん便利です。

まとめ　第8章：標準入出力ライブラリ

この章では、C言語のコンパイラが標準で用意している入出力のライブラリについて学習しました。いくつかの標準入出力ライブラリを利用してキーボードによる入力を取得する方法と画面への出力方法を学びました。

第8章で学習したこと

- キーボードからの入力や画面への出力をプログラムで扱うには、標準入出力ライブラリを利用する。
- scanf()関数を利用すると、変換指定子を使ってユーザーによるキーボードからの入力を、さまざまなデータ型に変換して受け取ることができる。
- scanf()関数は、スペース文字を含む入力を一括では受け取れない。
- gets()関数を利用すると、ユーザーによるキーボードからの入力を、すべて文字列変数に代入して受け取ることができる。
- printf()関数を利用すると、変換指定子を使ってさまざまなデータ型の変数を画面に出力することができる。
- puts()関数を利用すると、文字列データを画面に出力することができる。その際、行末に必ず改行が出力される。
- 文字列データを後から変更するには、sprintf()関数を利用する。

ステップアップ！

この章では、標準入出力ライブラリの基本的な扱い方を学習しました。C言語には、実はこれ以外にもたくさんの標準ライブラリが存在します。しかしこれらは、この章で学習した基礎を押さえておけば同じ要領で使えるものがほとんどです。この章の内容は、プログラミングでの応用のための基礎知識としてしっかりと理解しておきましょう。

問1 scanf()関数の利用

ユーザーに整数型のデータを入力させ、その数値を画面に表示してください。

答1

整数型データの入力を受け取るには、**scanf()** 関数を利用して変換指定子に「**%d**」を指定します。

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int n;
06      printf("整数を入力してください:");
07      scanf("%d", &n);
08      printf("¥nあなたの入力した整数は%dです。", n);
09      return 0;
10  }
```

問2 gets()関数の利用

ユーザーからの「My name is Suzuki Kenichi.」という文字列の入力を受け付け、入力された内容をそのまま表示してください。

答2

問の文字列にはスペース文字が含まれているので、**scanf()**関数では扱えません。スペース文字を含む文字列を受け取るには**gets()**関数を利用します。

```
01  #include <stdio.h>
02
03  int main()
04  {
05      char str[256];      /* 十分に大きな領域を確保する */
06      printf("自己紹介をしてください:");
07      gets(str);
08      printf("¥n■あなたの入力した自己紹介文¥n");
09      printf("%s", str);
10      return 0;
11  }
```

問3 puts()関数の利用

puts()関数を利用して、「コーヒーにミルクを入れますか?」と画面に表示してください。その際に、**char**型の配列に文字列を格納し、その配列を表示するようにしてください。

答3

puts()関数は、引数に文字列が格納された配列の先頭アドレスを指定します。解答例は次のようになります。

```
char str[] = "コーヒーにミルクを入れますか?";
puts(str);
```

第9章

Visual Learning Introduction of C

ファイル入出力

Section 31

ファイルポインタ

覚えておきたいキーワード

- ファイルポインタ
- fopen()関数
- fclose()関数

ワープロで作成した文書などをファイルに出力したり、ファイルからデータを読み込んで利用したりする機会も多いでしょう。C言語では、ファイルの入出力についても標準入出力を利用して行います。このセクションでは、ファイルを扱うための基礎を学習します。

1. ファイルを扱うためのしくみ

■ ファイルポインタとは…

ファイルの読み書きを行うためには、プログラムはさまざまな情報（ディスクのどこにアクセスしているかなど）を覚えておかなければなりません。ファイルの読み書きに関連した情報を1つにまとめて定義しているのが「**FILE**構造体」です。ファイルの読み書きを行うプログラムでは、「**FILE**構造体のポインタ」を使って1つのファイルを表します。これを「**ファイルポインタ**」といいます。**FILE**構造体のメンバについては詳細を知る必要はありません。「ファイルを扱うために利用する構造体」と認識してください。

図1 ファイルポインタ

ファイルポインタを宣言するには、次のように記述します。

構文　ファイルポインタの宣言

```
FILE *ポインタ名;
```

9 ファイル入出力

■ ファイルを扱う流れ

C言語では、ファイルを扱う際に、次の流れで操作します。

(1) ファイルをオープンする。

(2) ファイルへの読み書きを行う。

(3) ファイルをクローズする。

ファイルのオープンは「**fopen()**関数」で行い、開いたファイルに対してはファイルポインタを使って読み書きを行います。ファイルに対する処理が終わったら「**fclose()**関数(P.189参照)」を利用してファイルをクローズします。

2. ファイルのオープン

■ fopen()関数の書式

ファイルをオープンするには、「**fopen()**関数」を利用します。**fopen()**関数を利用するには、次の書式に従います。

構文 ファイルのオープン(fopen()関数)

```
FILE *ポインタ名 = fopen(ファイル名, オープンするモード);
```

次に、戻り値や引数について順に解説します。

■ fopen()関数の戻り値

FILE構造体は、変数を作成しません。**fopen()**関数の戻り値としてファイルポインタを受け取り、そのポインタを用いてファイルを操作します。**FILE**構造体の変数は**fopen()**関数を実行すると、関数内で自動的に確保されます。**fopen()**関数で確保されたメモリは、スコープ(P.112参照)に関係なくファイルが**fclose()**関数によってクローズされるまで有効な状態で保持されます。

なお、ファイルが何らかのエラー(ファイルが見つからないなど)によってオープンできない場合、戻り値として**NULL**(P.147参照)が返ります。

■ ファイル名の指定

fopen()関数の第1引数には、オープンしたいファイル名を文字列型で指定します。第2引数には、そのファイルをオープンするモードを指定します。

ファイルをオープンするモードには、次のようなものがあります。

表1 ファイルのオープンモード

引数	モード	ファイルが存在しない場合	ファイルが存在する場合
"r"	読み込み	エラー	－
"w"	書き込み	新規にファイルが作成されます。	ファイルの中身が失われます。
"a"	追加書き込み	新規にファイルが作成されます。	ファイルの末尾からデータを追加していきます。
"r+"	読み書き	エラー	－
"w+"	読み書き	新規にファイルが作成されます。	ファイルの中身が失われます。
"a+"	追加読み書き	新規にファイルが作成されます。	ファイルの末尾からデータを追加していきます。

また、第2引数には、目的のモードを文字列型で指定します。たとえば「test.txt」というファイルを「読み込みモード」でオープンしたい場合は、

```
FILE *fp = fopen("test.txt", "r");
```

のように記述します。

■ ファイルの[illegible]

ファイルには、次の2つの種類があります。

(1) テキストファイル

テキストデータのみで構成されたファイルのことを指します。たとえばC言語のソースファイルや、拡張子「.txt」で表されるファイルのことです。

(2) バイナリファイル

テキストファイル以外のファイルのことを指します。絵や音のデータなど、テキストデータではないデータを保存しているファイルの総称です。

ファイルを開く際には、「**テキストモード**」で開くか「**バイナリモード**」で開くかを選択することができます。指定を省略すると、バイナリモードで開かれます。

ファイルの種類を指定するには、**fopen()**関数の第2引数に次の文字を加えて指定します。

表2 追加文字

追加文字	モード	機　能
"b"	バイナリ	ファイルのデータをそのまま読み込みます。
"t"	テキスト	テキストファイル上の「改行」コードを、プログラムで利用できる「¥n」に変換してデータを読み込みます。

たとえば、「test.txt」というファイルを「テキストモード」かつ「読み込みモード」で開くには、次のように記述します。

```
FILE *fp = fopen("test.txt", "rt");
```

3. ファイルのクローズ

■ fclose()関数の書式

ファイルに対しての処理が終了したら、「**fclose()**関数」を実行します。**fclose()**関数が実行されると、**fopen()**関数によって確保された**FILE**構造体のメモリ領域が解放されます。

fopen()関数で確保したメモリ領域は、**fclose()**関数が実行されるまでずっと確保されたままです。**fclose()**関数が呼び出されないままプログラムが終了すると、メモリを確保したままになる可能性があるので、**fopen()**を実行した場合は、最後に必ず**fclose()**関数を実行するようにします。

fclose()関数を利用するには、次の書式に従います。

構文 ファイルのクローズ(fclose()関数)

```
fclose(ファイルポインタ);
```

たとえば**fopen()**関数と**fclose()**関数を利用したプログラムは、次のようになります。

```
FILE *fp = fopen("test.txt", "rt");
⋮
fclose(fp);
```

fopen()関数が返した
ファイルポインタを指定します。

fclose()関数を実行すると、**fopen()**関数によって確保されていた**FILE**構造体のメモリ領域が無効になります。**fclose()**を実行した後は、ファイルポインタの参照先のアドレスは無効になるため、このポインタを使うことはできません。

値が無効になったファイルポインタや、オープンに失敗したファイルポインタを利用しようとすると、不正なメモリアクセスとしてエラーになる可能性があります。

Section 32

テキストファイルの読み書き

覚えておきたいキーワード
- fgets()関数
- fputs()関数
- fprintf()関数

ファイルポインタと、ファイルのオープン、クローズが理解できたら、次にファイルの読み書きを行いましょう。テキストファイルを利用すれば、文字列のデータを保存することができます。また、ファイルから文字列の配列にデータを読み込むこともできます。

1. テキストファイルの読み込み

■ fgets()関数の利用

テキストファイルを読み込むには、ファイルを読み込み可能な状態でオープンした後、「**fgets()**関数」を利用します。**fgets()**関数を利用するには、次のように記述します。

構文 テキストファイルからの読み込み（fgets()関数）

```
char *ポインタ名 = fgets(char型配列, 読み込む文字数, ファイルポインタ);
```

fgets()関数の戻り値は**char ***型で受け取ります。**fgets()**関数内でエラーが発生したり、ファイルの終端まで読み込んでいた場合には**NULL**が返ります。

fgets()関数でファイルから読み込んだテキストデータは、第1引数の「**char**型配列」に格納されます。読み込む文字数は第2引数の「読み込む文字数」で設定します。第1引数には、第2引数で指定した文字数よりも長い配列を指定しなければなりません。第3引数には、**fopen()**関数でオープンしたファイルポインタを指定します。

fgets()関数を利用したプログラムは、次のようになります。

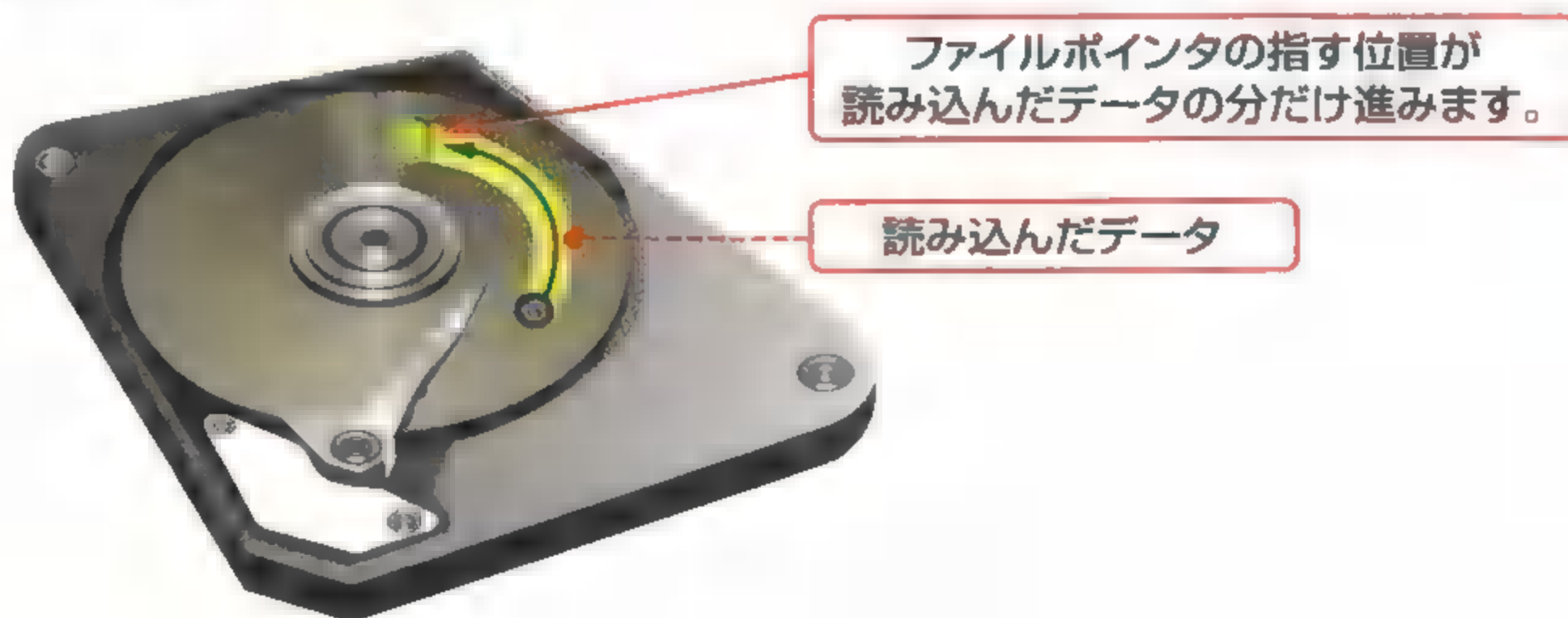
```
FILE *fp;
char str[256];
fp = fopen("test.txt", "rt");
char *pread = fgets(str, 256, fp);
fclose(fp);
```

test.txtを読み込みモードでオープンします。

strの配列長が256なので、読み込む文字数も最大256と設定します。

なお、ファイルポインタが指し示すディスク上の位置は、読み込んだデータ分だけ先に進んでいきます。一度ファイルを読み込むとファイルポインタの位置が動いてしまい、再び同じデータを読み込むことができなくなるので注意してください。どうしても同じデータを2回読み込みたい場合は、**fseek()**関数（P.197参照）を利用します。

図1 ファイルからの読み込み

■ fgets()関数での注意点

fgets()関数を利用する際、テキストファイルを一度に全部は読み込めない場合があります。テキストファイル上に次のデータがある場合、データの読み込みは中断されます。

(1) 改行

(2) 指定した長さ分のデータ

(3) ファイルの終端

ファイルポインタは、読み込みが中断された位置を指します。そのため、再び**fgets()**関数を実行すると、中断された位置から読み込みが再開されます。

fgets()関数でファイルを最後まで読み込むには、ファイルの終端が読み込まれるまで**fgets()**関数を繰り返し実行します。**fgets()**関数は、ファイルポインタが終端にある状態で実行すると、戻り値として**NULL**を返します。つまり、**fgets()**関数の戻り値が**NULL**でない間だけ処理を繰り返せば、ファイルの終端までのデータをすべて読み込むことができます。

fgets()関数の戻り値がNULLかどうかを調べるには、次のように記述します。

```
char *ポインタ名 = fgets(char型配列, 読み込む文字数, ファイルポインタ);
if( ポインタ名 == NULL ) {
    ⋮
```

fgets()関数の戻り値がNULLかどうかを調べます。

このように、あるポインタがNULLかどうかを調べることを「NULLチェック」などといいます。なお、fgets()関数と同様に、fopen()関数を呼び出す際も戻り値のNULLチェックを行います。fopen()関数の戻り値の型は「FILE *」型ですが、fgets()関数の場合と同じようにチェックすることができます。

```
FILE *fp = fopen("test.txt", "rt");
if( fp == NULL ) {
    ⋮
```

FILE *型であっても、if文でNULLチェックを行うことができます。

fgets()関数の戻り値をNULLチェックするプログラムは、次のようになります。

Sample0901.c　fgets()関数の注意点

```
01 #include <stdio.h>
02
03 int main()
04 {
05     char *pread;
06     char str[256];
07
08     FILE *fp = fopen("test.txt", "rt");
09     if(fp == NULL) {
10         printf("ファイルがオープンできません¥n");
11         return 0;
12     }
13     while( 1 ) {
14         pread = fgets(str, 10, fp);
```

ファイルがオープンできない場合、main()関数を抜けてプログラムを終了させます。

ファイルの最後を読み込むまで、処理を繰り返します。

ファイルから最大で10文字分読み込みます。

```
15            if( pread == NULL ){
16                printf("ファイル終端¥n");
17                break;
18            }
19            printf("[%s]", str);
20        }
21        fclose(fp);
22        printf("処理終了");
23
24        return 0;
25    }
```

ファイルの終端まで読み込むと、preadにはNULLが返ります。

ファイルの終端まで読み込んだ場合は、ループ処理を抜けます。

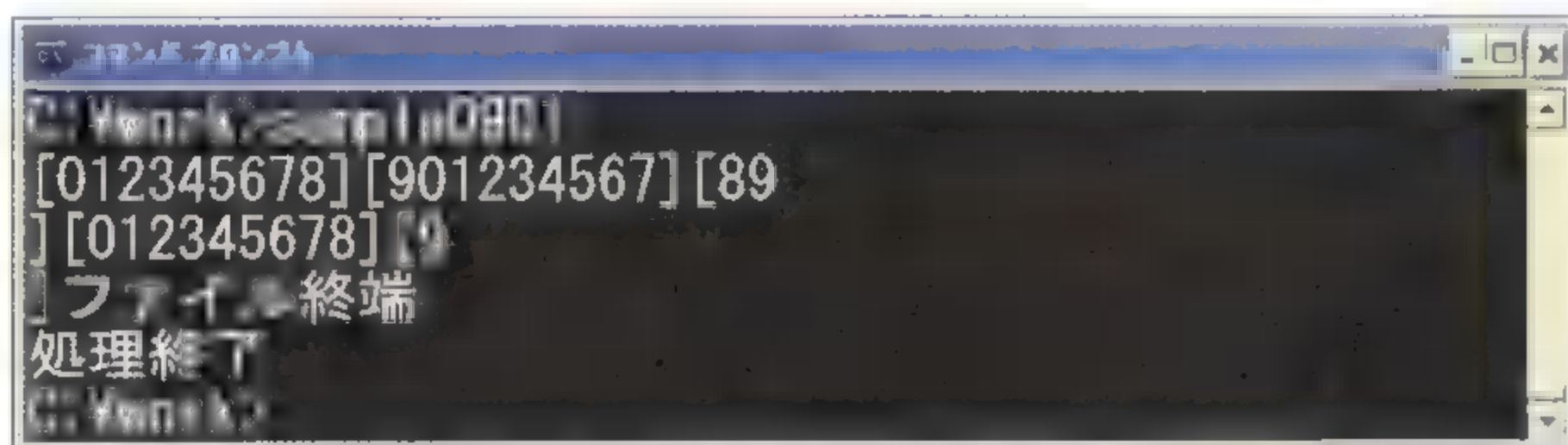
読み込んだ文字列を表示します。

では、実際にプログラムを実行してみましょう。「test.txt」という名前で次のファイルを用意し、プログラムと同じディレクトリに置きます。

テキストファイルの内容

```
01234567890123456789↵
0123456789↵
```

すると、実行画面は次のようになります。

```
C:¥work>sample0801
[012345678][901234567][89
][012345678][9
]ファイル終端
処理終了
C:¥work>
```

13行目の**while**文は、常に条件が「真」になる無限ループ処理です。ファイルの終端を読み込むまではこの**while**文によって何度でも繰り返し読み込み処理が実行されます。

14行目では、ファイルから10文字ずつ読み込んでいます。15行目は、14行目の**fgets()**関数がデータを読み込んだかどうかをチェックしています。ファイルの終端まで読み込んでいる状態で**fgets()**関数を実行すると、**pread**には**NULL**が返されるため、**if**文の条件が真になります。

16〜17行目は、ファイルの終端まで読み込んだ場合に実行される処理です。この例では、ファイルの終端まで読み込んだことを表示し、**break**文でループを抜けます。

19行目では、**fgets()**関数で読み込んだ文字列を[]で囲んで表示しています。これは、**fgets()**関数で読み込んだ文字列が、どこからどこまでなのかをわかりやすく示すために表示しています。

実行結果を見てみると、10文字ずつ読み込むよう指定しているのに、実際は9文字しか読み込まれていません。これは、文字列の終端にNULL文字（**'¥0'**）を挿入するため、**fgets()**関数がその1文字分を残して読み込みをやめるからです。

また、改行を読み込むと、そこで文字列が切れることもわかります。ファイルポインタが終端まで進んだ後に**fgets()**関数を利用すると、15行目の**if**文の条件が真になりループから抜けます。

2. テキストファイルの書き込み

■ fputs()関数の利用

文字列データをファイルに書き込むには、ファイルを書き込み可能な状態でオープンして、「**fputs()**関数」を利用します。**fputs()**関数を利用するには、次のように記述します。

構文　テキストファイルの書き込み（fputs()関数）

```
fputs(出力する文字列, ファイルポインタ);
```

fputs()関数は、第2引数のファイルポインタが指しているファイルに、第1引数の文字列を出力します。出力する文字列の終端には、NULL文字（**'¥0'**）を入れなければいけません。

fputs()関数は、**fgets()**関数と違って文字列を最後まで出力します。文字列の途中に改行などが入っていても、書き込み処理は中断しません。また、**puts()**関数のように、文字列の最後に自動的に改行（**'¥n'**）を入れることもありません。

fputs()関数を利用したプログラムは、たとえば次のようになります。

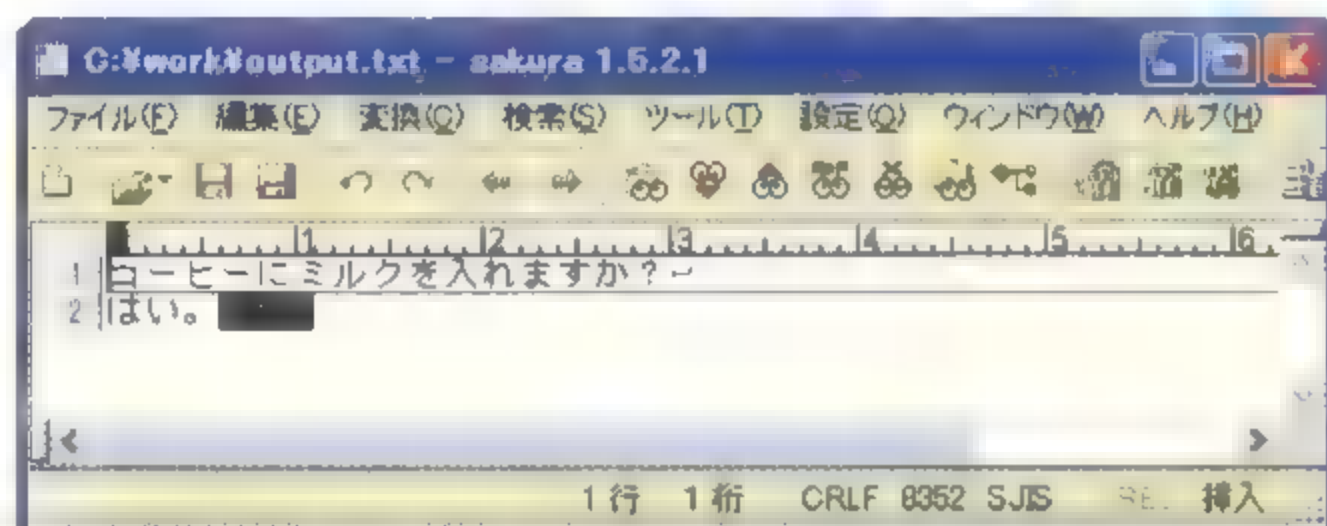
Sample0902.c fputs()関数の利用

```
01  #include <stdio.h>
02
03  int main()
04  {
05      char str[] = "コーヒーにミルクを入れますか?¥nはい。";
06      FILE *fp = fopen("output.txt", "wt");
07      if(fp == NULL) {
08          printf("ファイルがオープンできません¥n");
09          return -1;
10      }
11      fputs(str, fp);
12      fclose(fp);
13      printf("処理終了");
14
15      return 0;
16  }
```

ファイルに出力する文字列を設定します。

output.txtを書き込みモードでオープンします。

文字列strの内容をfpに出力します。

出力されるテキストは次のようになります。

なお、ファイルをクローズするまでの間であれば、続けて**fputs()**関数を実行することで、連続したデータを出力することができます。

たとえば、**fputs()**関数を次のように記述した場合、

```
fputs("コーヒーにミルクを入れますか?¥n");
fputs("はい、入れます。¥n");
```

出力されるテキストは、次のようになります。

```
コーヒーにミルクを入れますか?
はい、入れます。
```

fprintf()関数の利用

整数や浮動小数点数など、文字列以外のデータをテキストファイルに書き込む場合には、「**fprintf()**関数」を利用します。**fputs()**関数では、文字列データしか扱うことができませんが、**fprintf()**関数では変換指定子を利用することにより、さまざまな形式のデータを扱うことができます。

fprintf()関数でファイルに出力するには、書き込み可能な状態でファイルをオープンし、次のように記述します。

構文 テキストファイルの書き込み(fprintf()関数)

```
fprintf(ファイルポインタ, "……変換指定子……", 変数1, 変数2, ……);
```

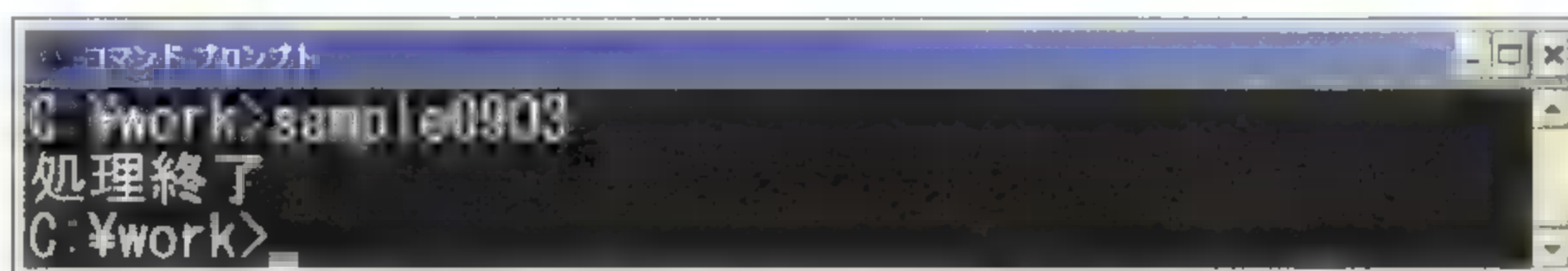
第1引数にはファイルポインタを指定します。第2引数と第3引数は**printf()**関数と同じ方法で記述します。**printf()**関数では画面に出力されますが、**fprintf()**関数ではファイルポインタに出力されます。

fprintf()関数を利用したプログラムは、次のようになります。

Sample0903.c fprintf()関数の利用

```
01 #include <stdio.h>
02 
03 int main()
04 {
05     FILE *fp = fopen("output.txt", "wt");
06     if(fp == NULL) {
07         printf("ファイルがオープンできません¥n");
08         return -1;
09     }
10     fprintf(fp, "コーヒー %d ml¥n", 200);
11     fprintf(fp, "ケーキ %d 個¥n", 2);
```

1 このファイルポインタに対して、
2 このデータを書き込みます。

```
12
13      fclose(fp);
14      printf("処理終了");
15
16      return 0;
17  }
```

出力されるテキストは、次のようになります。

Column

ファイルを開き、データを読み込んだ場合、ファイルポインタが読み込んだデータの長さ分だけ進んでしまい、同じデータを読み込むことができなくなります。このような場合、**fseek()**関数を利用してファイルポインタを目的の場所に動かすことができます。

構文 fseek()関数

```
fseek(ファイルポインタ, 移動させるバイト数, 初期位置);
```

「ファイルポインタ」には、すでに開いているファイルポインタを指定します。「初期位置」には、次の3つのうちいずれかの値を記述します。

(1)**SEEK_SET**(ファイルの先頭)
(2)**SEEK_END**(ファイルの終端)
(3)**SEEK_CUR**(ファイルポインタの現在位置)

なお、これらは定数で、**stdio.h**ファイルに定義されています。
「移動させるバイト数」には、「初期位置」に指定した場所から何バイト動かしたいかを記述します。たとえば、ファイルの先頭に戻りたい場合は、次のように記述します。

```
fseek(fp, 0, SEEK_SET);
```

「ファイルの先頭」から「0バイト」の場所に移動させるという意味です。

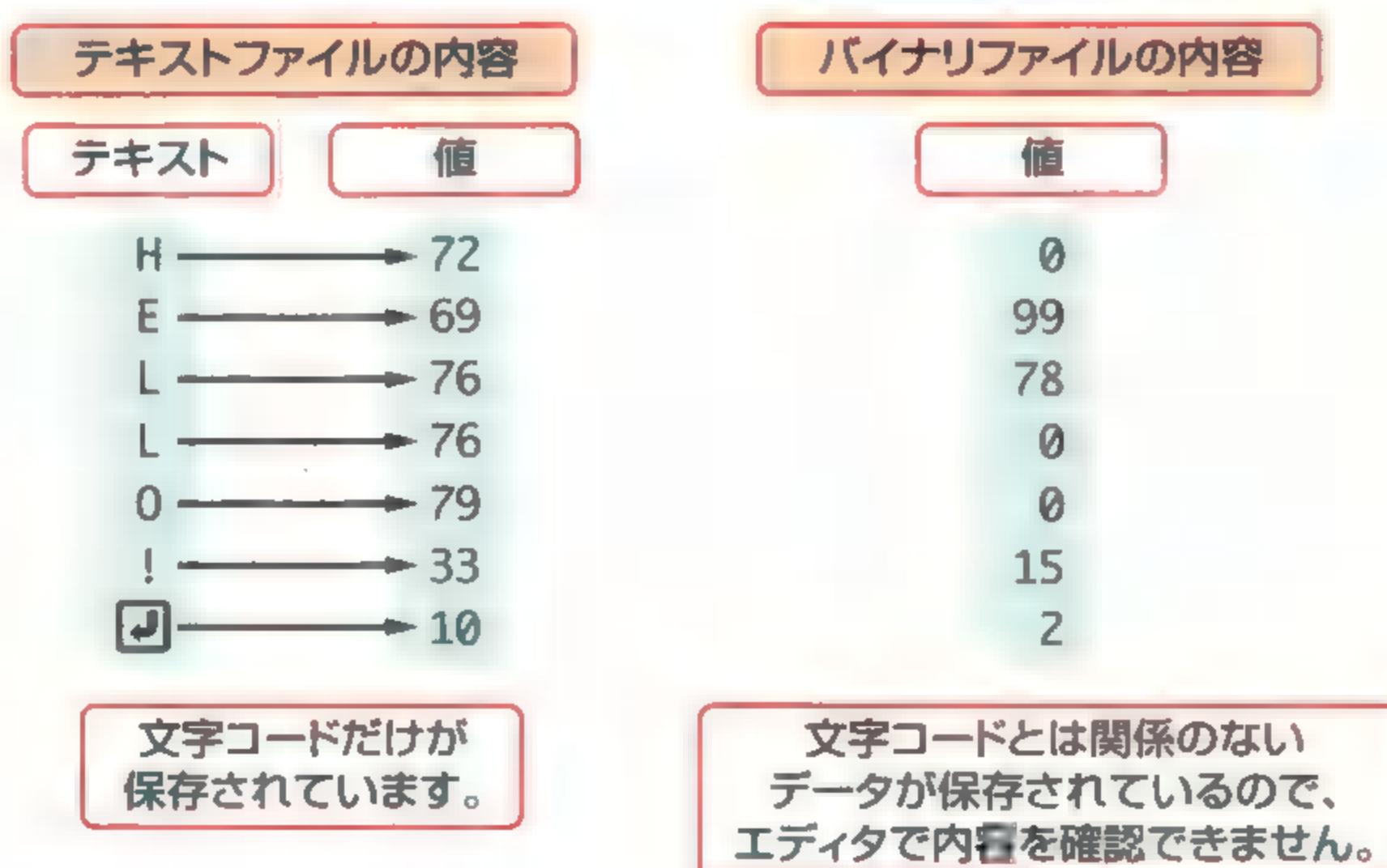
バイナリファイルの読み書き

テキストファイルには、文字のデータしか保存することができません。プログラムで演算を行った結果の値など、文字以外のデータを保存する場合はバイナリファイルを利用します。このセクションでは、バイナリファイルにデータを読み書きする方法を学習します。

1. バイナリファイルとは…

バイナリファイルとは、画像データや音声データなど、テキスト以外のデータが保存されたファイルのことです。テキストファイルは文字コードだけで構成されたファイルを指し、バイナリファイルはそれ以外のデータが保存されたファイルを指します。

バイナリファイルの内容は、テキストエディタで確認することができません。

図1 テキストファイルとの違い

2. バイナリファイルの書き込み

バイナリファイルに書き込むには、まずバイナリモードで書き込み可能となるようにファイルをオープンします。

```
FILE *fp = fopen("data.bin", "wb");
```

第2引数に"b"を付けることで、
バイナリモードでファイルをオープンします。

次に、データを書き込むために「**fwrite()**関数」を利用します。これまでの**fputs()**関数や**fprintf()**関数と比較すると、**fwrite()**関数は少々複雑です。**fwrite()**関数の考え方は「1つの要素が何バイトの配列から、何個の要素をファイルに書き込むか」を指定するというものです。

構文 fwrite()関数（配列の書き込み）

```
fwrite(書き込みデータを含む配列,
        配列の1つの要素のバイト数,
        書き込む要素の数, ファイルポインタ);
```

また、配列ではないデータをバイナリモードで書き込みたい場合は、次のように記述します。

構文 fwrite()関数（変数の書き込み）

```
fwrite(書き込むデータのアドレス,
        書き込むデータのバイト数, 1, ファイルポインタ);
```

これは、つまり「長さが1の配列を保存する」のと同じ意味です。

fwrite()関数を利用したプログラムは、次のようになります。

Sample0904.c バイナリファイルの書き込み

```
01  #include <stdio.h>
02
03  int main()
04  {
05      /* 書き込みたいデータ */
06      int out[10] = {
07          1,2,3,4,5,6,7,8,9,10
08      };
09
10      FILE *fp = fopen("data.bin", "wb");
11      if(fp == NULL) {
12          printf("ファイルがオープンできません¥n");
```

バイナリモードで書き込み用に
ファイルをオープンします。

```
13        return -1;
14     }
15
16     fwrite(out, sizeof(int), 10, fp);
17
18     fclose(fp);
19     printf("処理終了");
20
21     return 0;
22 }
```

int型のサイズの要素を10個分書き込みます。

これは、配列**out**のデータをファイル「**data.bin**」に書き込むプログラムです。16行目で**fwrite()**関数を実行して、配列**out**の内容をファイルに書き込みますが、このとき関数の第2引数に**sizeof**演算子(詳細はP.169参照)を利用して**int**型のバイト数を設定しています。Windowsでプログラミングを行う場合、**int**型は「4バイト」なのですが、C言語でプログラミングを行う場合の「お約束」として、このように記述します。これは「このプログラムを違う環境に持っていっても動作する」というC言語の本来のメリットを生かすためです。また構造体を利用する場合、わざわざバイト数を数えるのは面倒なので**sizeof**演算子を利用します。

第3引数では、配列の長さを指定しています。**fwrite()**関数に慣れたら、この部分も次のように記述するとよいでしょう。

```
fwrite(out, sizeof(int), (sizeof(out)/sizeof(int)), fp);
```

配列の長さを「配列のバイト数÷1つの配列要素のバイト数」で算出します。

このように記述すると、配列の長さが変わった場合でも**fwrite()**関数の呼び出しを手直しする必要がなくなります。

3. バイナリファイルの読み込み

バイナリファイルを読み込むには、バイナリモードで読み込み可能となるようにファイルをオープンして「**fread()**関数」を利用します。

fread()関数の使い方は、**fwrite()**関数に似ています。違いは、**fwrite()**関数では第1引数が「書き込みたいデータ」であるのに対し、**fread()**関数が「読み込んだデータを保存するバッファ」であることです。

構文 fread()関数（配列への読み込み）

```
fread(読み込むデータの格納先の配列,
      配列の1つの要素のバイト数,
      読み込む要素の数, ファイルポインタ);
```

読み込むデータの格納先の配列には先頭要素のアドレスを渡します。

また、読み込んだデータを保存するバッファが配列でない場合は、**fwrite()**関数の場合と同様に次のように記述します。

構文 fread()関数（変数への読み込み）

```
fread(読み込むデータの格納先のアドレス,
      読み込むバイト数, 1, ファイルポインタ);
```

fread()関数を利用して、先ほど**fwrite()**関数で作成した「data.bin」ファイルを読み込むプログラムを作成してみましょう。

Sample0903.c バイナリファイルの読み込み

```
01  #include <stdio.h>
02
03  int main()
04  {
05      /* データを読み込むバッファ */
06      int in[10];
07      int i;
08
09      FILE *fp = fopen("data.bin", "rb");
```

バイナリモードで読み込み用にファイルをオープンします。

```
10      if(fp == NULL) {
11          printf("ファイルがオープンできません¥n");
12          return -1;
13      }
14
15      fread(in, sizeof(int), 10, fp);
16
17      for(i=0; i<10; i++) {
18          printf("%2d番目:%2d¥n", i, in[i]);
19      }
20
21      fclose(fp);
22      printf("処理終了");
23
24      return 0;
25  }
```

int型のデータを配列の長さ分だけ読み込みます。

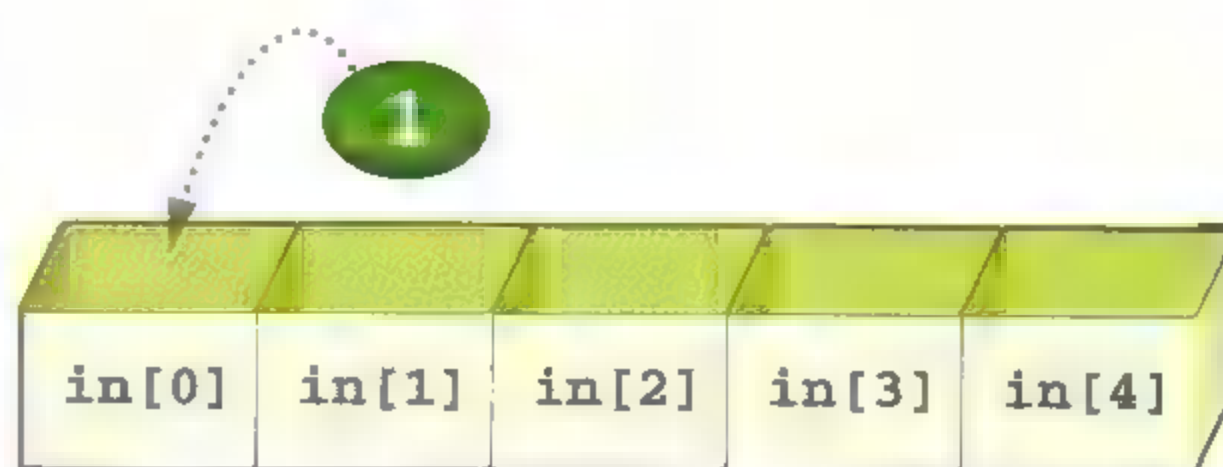
15行目の**fread()**関数では、ファイルから**int**型のデータを10個読み込んでいます。

どのようにデータを読み込んでいるかを少し細かく見てみましょう。ファイルから最初に「**sizeof(int)**」バイトのデータを読み込みます。これにより、ファイルの先頭4バイトのデータ、つまりそこに保存されている「1」という数値が読み込まれます。これを配列「**in**」の0番目にコピーします。

バイナリファイルの内容

```
01 00 00 00 02 00 00 00
03 00 00 00 04 00 00 00
```

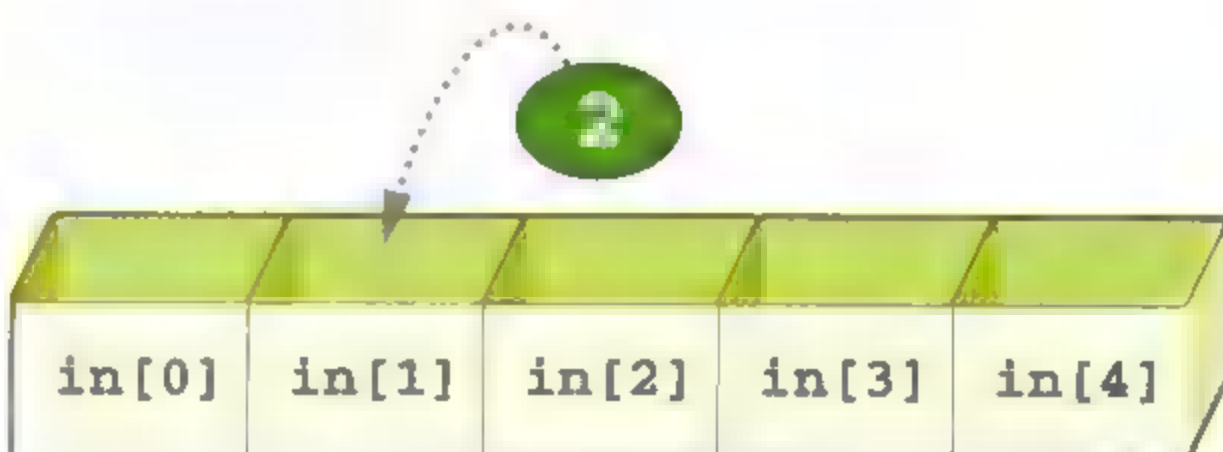
1

in[0]	in[1]	in[2]	in[3]	in[4]

1 ここから4バイトをまず読み込みます。

2 読み込んだ4バイトを格納先のアドレスにコピーします。

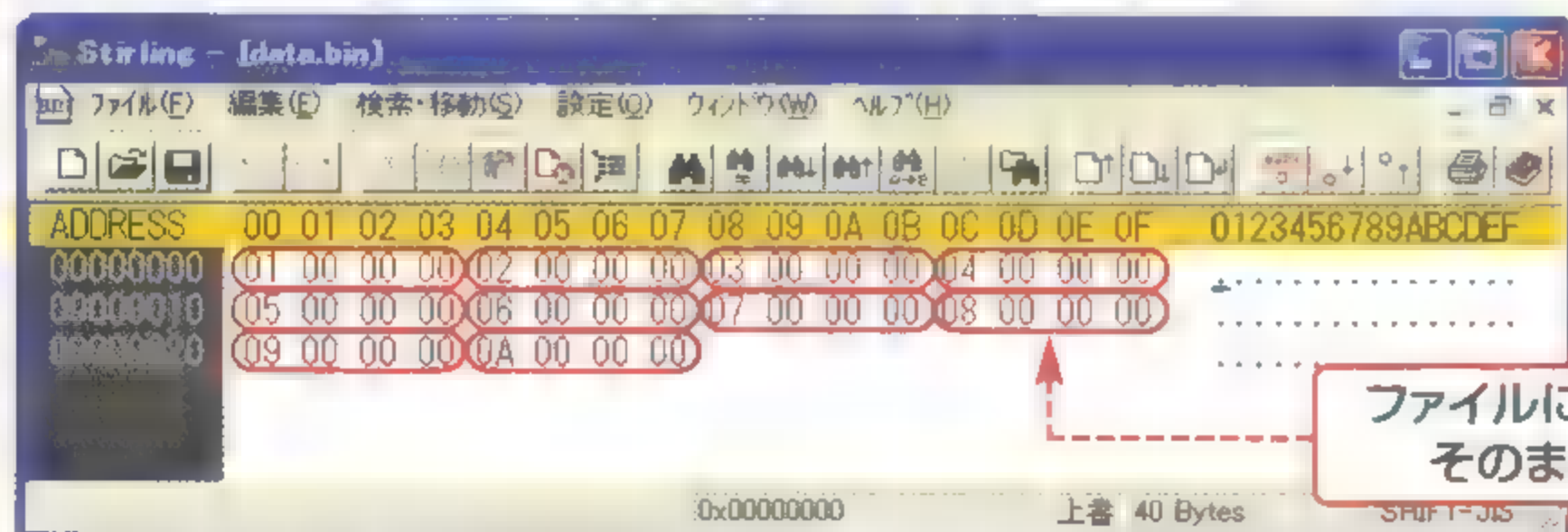
4バイトを読み込むと、ファイルポインタの指す位置が4バイトだけ進みます。次の4バイトを読み込むと、そこに保存されている「2」のデータを、配列**in**の1番目にコピーします。この読み込みの動作を10回繰り返すと、**fread()**関数は処理を終了します。

バイナリファイルの内容

```
01 00 00 00 02 00 00 00
03 00 00 00 04 00 00 00
```

1 ここから4バイトを読み込みます。

2 読み込んだ4バイトを格納先のアドレスにコピーします。

Column バイナリエディタ

本文中では利用しませんでしたが、バイナリモードで保存したデータを表示、編集するためのツールとして「バイナリエディタ」があります。Windows上で動作する代表的なバイナリエディタには「Stirling」や「Quick Be」などがあります。「Stirling」で、本文中で作成した「data.bin」を開くと、次のように表示されます。

なお、ファイルに値を保存した場合も、数値は「リトルエンディアン(P.166参照)」で表現されます。

まとめ 第9章: ファイル入出力

この章では、プログラムからファイルを読み込んだり、書き込んだりする手順を学習しました。また、テキストファイルとバイナリファイルの違いや、利用するファイル入出力の関数の違いについてもあわせて学びました。

第9章で学習したこと

- プログラムからファイルを扱うには、必ずファイルポインタを通して操作を行う。
- ファイルを扱うには、まずfopen()関数でファイルを「オープン」して、ファイルを使い終わったら必ずfclose()関数で「クローズ」する。
- fopen()関数の戻り値がファイルポインタとなる。戻り値がエラーを示す場合はファイルが開けないため、処理を中断する。
- ファイルポインタは、次にファイルの読み込みが指示されたときに読み込み始める場所を指している。
- ファイルを読み込むと、ファイルポインタが指している読み込み位置は、読み込んだデータのサイズ分だけ進む。
- fseek()関数を利用すれば、ファイルポインタの指しているファイルの位置を動かすことができる。

ステップアップ!

ファイルの取り扱いは、必ずファイルポインタを通して行うことから、やや特殊な感じを受けます。ファイルポインタへの入出力は、常にファイルポインタが指し示す先に対して行われるという原則があるので、それを頭に入れておくようにします。読み込んだり、書き込んだりすると、ファイルポインタは入出力した分だけ前に進んでいくことも重要なことなので覚えておきましょう。

バイナリファイルの場合、何バイトの情報をどういった順番で書き込んだかを把握しておく必要があります。そうでないと、読み込んだ際に元のデータが何バイトであったかがわからず、結果としてへんてこな値を読み込んでしまうことになります。書き込んだ順に書き込んだバイト数だけきちんと読み込むようにしましょう。

問1　もっとも基本的なプログラム

次の文章をテキストファイルとして保存するプログラムを作成してください。

あめんぼ赤いな、あいうえお⏎
今度はかならず、かきくけこ⏎
おはようサマンサ、さしすせそ⏎

答1

解答例として、テキストデータを3つの**char**型の配列として用意しました。1つの配列でテキストデータを用意して**fputs()**関数などで書き込んでも正解です。

```
FILE *fp;
char str1[] = "あめんぼ赤いな、あいうえお";
char str2[] = "今度はかならず、かきくけこ";
char str3[] = "おはようサマンサ、さしすせそ";

fp = fopen("test.txt", "wt");
fprintf(fp, "%s¥n%s¥n%s¥n", str1, str2, str3);
fclose(fp);
```

問2　バイナリファイルの読み書き

次のデータを書き込むプログラムと、読み込んで表示するプログラムを別々に作成してください。

バイト	数　値
4	10
4	200
1	'A'
1	'Z'

答2

まず、バイナリファイルを書き込むプログラムは、次のようになります。

```
FILE *fp;
int outnum[] = { 10, 200 };
char outch[] = { 'A', 'Z' };

fp = fopen("output.bin", "wb");
/* int型配列の書き込み */
fwrite(outnum, sizeof(int), 2, fp);
/* char型配列の書き込み */
fwrite(outch, sizeof(char), 2, fp);
fclose(fp);
```

次に、バイナリファイルを読み込むプログラムは、次のようになります。

```
FILE *fp;
int innum[2];
char inch[2];

fp = fopen("output.bin", "rb");
/* int型配列の読み込み */
fread(innum, sizeof(int), 2, fp);
/* char型配列の読み込み */
fread(inch, sizeof(char), 2, fp);
fclose(fp);

printf("%d, %d, %c, %c",
        innum[0], innum[1], inch[0], inch[1]);
```

第 10 章

Visual Learning Introduction of C

プリプロセッサ命令

ヘッダーファイルの取り込み

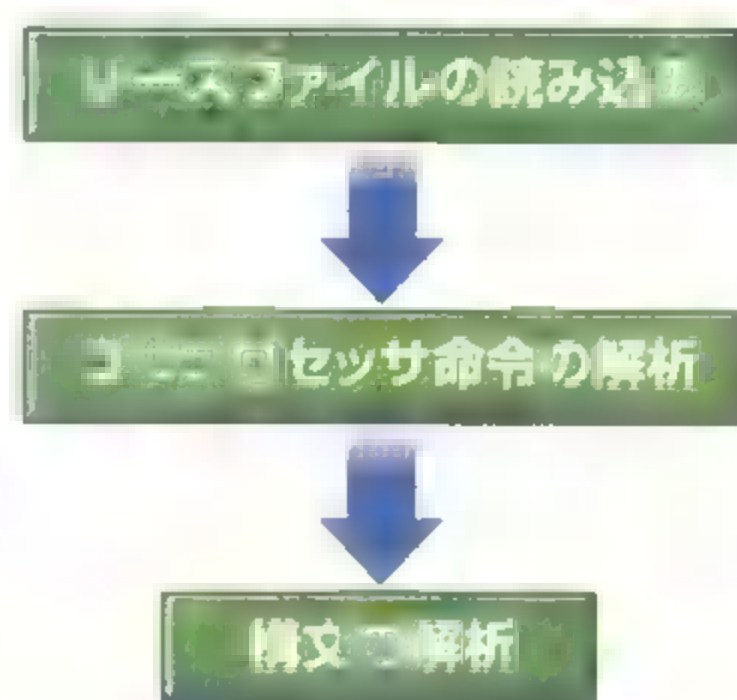
覚えておきたいキーワード
- ヘッダーファイル
- #include
- stdio.h

コンパイルを行う前に処理すべきことを記述するには、プリプロセッサ命令を利用します。たとえば、標準ライブラリの関数を利用するためには、プリプロセッサ命令のひとつであるinclude文を利用して、stdio.hファイルを読み込むことが必要になります。

1. プリプロセッサ命令とは...

「**プリプロセッサ命令**」とは、コンパイルを行うための準備の記述です。たとえば、今までプログラム例の先頭に必ず記述してきた「**#include**」も、プリプロセッサ命令のひとつです。

プリプロセッサ命令は、コンパイラがソースファイルの文法チェックを行う前に、あらかじめ処理されます。プリプロセッサ命令により、ソースファイルをコンパイルするときに利用する外部ファイル(「**ヘッダーファイル**」といいます)を指定したり、ソースファイル内で利用する定数を定義しておいたりすることができます。

図1　コンパイルの流れ

なおプリプロセッサ命令は、命令文の先頭に「**#**」を付けて利用します。「**#include**」の他には「**#define**(P.214参照)」などがありますが、これらの命令の末尾には「**;**(セミコロン)」を付けません。

```
#include <stdio.h>
```

行末に「;」を付けません。

2. 別のソースファイルにある関数の利用

■ include文とは…

今までプログラム例の一番上に念仏のように記述していた「**#include <stdio.h>**」という文ですが、この文は「**stdio.h**の内容を取り込む」という意味です。これを「**include**文」と呼びます。

「**stdio.h**」はC言語が標準で備えているヘッダーファイルで、中には**printf()**関数や**fopen()**関数などのプロトタイプ宣言（P.116参照）が記述されています。ソースコードでこれらの関数を利用できるのは、**stdio.h**にプロトタイプ宣言が記述されているからです。

「ヘッダーファイル」とは、関数のプロトタイプ宣言や構造体の型宣言などをまとめたファイルのことです。一般的に、ヘッダーファイルでは変数の宣言や、関数の定義などは行いません。

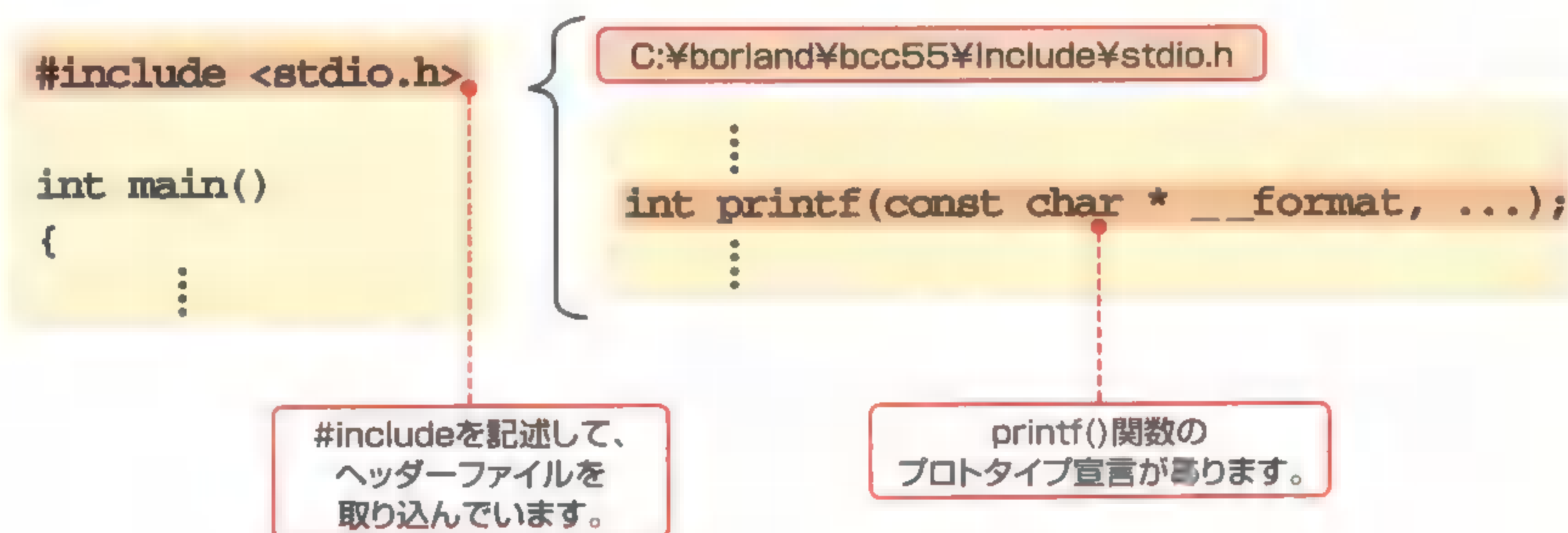



図2 include文

stdio.hに限らず、C言語では数多くのヘッダーファイルを標準で備えています。そういったヘッダーファイルを**include**文により利用するには次のように記述します。

構文 include文（標準ライブラリ）

```
#include <ヘッダーファイル名>
```

行末に「**;**（セミコロン）」が付かないことに注意してください。**include**文の後に続けて別の文を記述することはできません。

■ ヘッダーファイルの作成

ヘッダーファイルを自分で作成することもできます。ヘッダーファイルには、他のソースファイルから扱いたい情報、たとえば、関数のプロトタイプ宣言などを記述します。

ヘッダーファイルを利用して、他のソースファイルの関数を呼び出すには次に示すような手順に従います。まず3つのファイルを同じフォルダに用意してください。

(1) main()関数のあるSample1001.c（ソースファイル）

(2) 呼び出される関数を定義するdisp.c（ソースファイル）

(3) 呼び出される関数のプロトタイプ宣言を行うdisp.h（ヘッダーファイル）

■ 呼び出される関数の定義

最初に、**main()**関数から呼び出される関数の定義を行います。次のようなソースファイルを作成し、これをdisp.cという名前で保存します。

disp.c　呼び出される関数の定義

```
01  #include <stdio.h>
02
03  void disp(void)
04  {
05      printf("disp()関数を実行しました。¥n");
06  }
```

次に、disp.cに定義した関数のプロトタイプ宣言をヘッダーファイルに記述し、これをdisp.hという名前で保存します。

disp.h　ヘッダーファイルでのプロトタイプ宣言

```
01  void disp(void);
```

プロトタイプ宣言を行うだけなので、「#include <stdio.h>」は不要です。

最後に、**main()**関数を定義するソースファイルを作成します。disp.cで定義した**disp()**関数を利用するためにはdisp.hを**include**文で取り込むだけですみます。コンパイラは、プロトタイプ宣言を見つけると「関数の定義はコンパイルするソースファイルのどこかにあるはず」と考えて、自動的に探し出してくれます。

自分で作成したヘッダーファイルを**include**文で取り込むには、次のように記述します。ヘ

ッダーファイル名を「"(ダブルクォーテーション)」で囲むことに注意してください。

構文 include文(作成したヘッダー)

```
#include "ヘッダーファイル名"
```

次のようにmain()関数を定義したソースファイルを作成し、これをSample1001.cという名前で保存します。

Sample1001.c ヘッダーファイルの利用

```
01 #include <stdio.h>
02 #include "disp.h"
03
04 int main()
05 {
06     printf("main()関数を実行します。¥n");
07     disp();
08
09     return 0;
10 }
```

自分で作成したヘッダーをinclude文で取り込む場合は「"」で囲みます。

これらのファイルを用意したら、コンパイルを行います。

■ 複数のソースファイルのコンパイル

Borland C++ Compiler 5.5で複数のファイルをコンパイルするには、コマンドプロンプト上で「bcc32」に続いて、コンパイルしたいソースファイル名をスペースで区切って並べて入力します。ただし、ヘッダーファイル名は入力しません。

たとえば、Sample1001.cとdisp.cをコンパイルしたい場合、コマンドプロンプト上で次のように入力します。

これを実行すると、2つのソースファイルが続けてコンパイルされます。

```
bcc32 Sample1001.c disp.c
```

ソースファイル名はスペースで区切ります。

```
C:¥work>bcc32 sample1001.c disp.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
sample1001.c:
disp.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

C:¥work>
```

続いてプログラムを実行してみましょう。**main()**関数から**disp()**関数が呼び出されていることがわかります。

なお、関数のプロトタイプ宣言と同様に、ヘッダーに構造体を定義することもできます。

Column staticとextern

複数のソースファイルを扱う場合に、よく利用される2つのキーワードを紹介します。

extern

「**extern**」は、他のソースファイルで宣言したグローバル変数を利用したい場合に利用します。**extern**を利用するには、次のように記述します。

構文 extern

```
extern データ型 変数名;
```

externを利用したプログラムは、次のようになります。

lib.c グローバル変数の宣言

```
01  #include <stdio.h>
02
03  int global;
04
05  void dispGlobal(void)
06  {
07      printf("global = %d¥n", global);
08  }
```

グローバル変数を宣言しています。

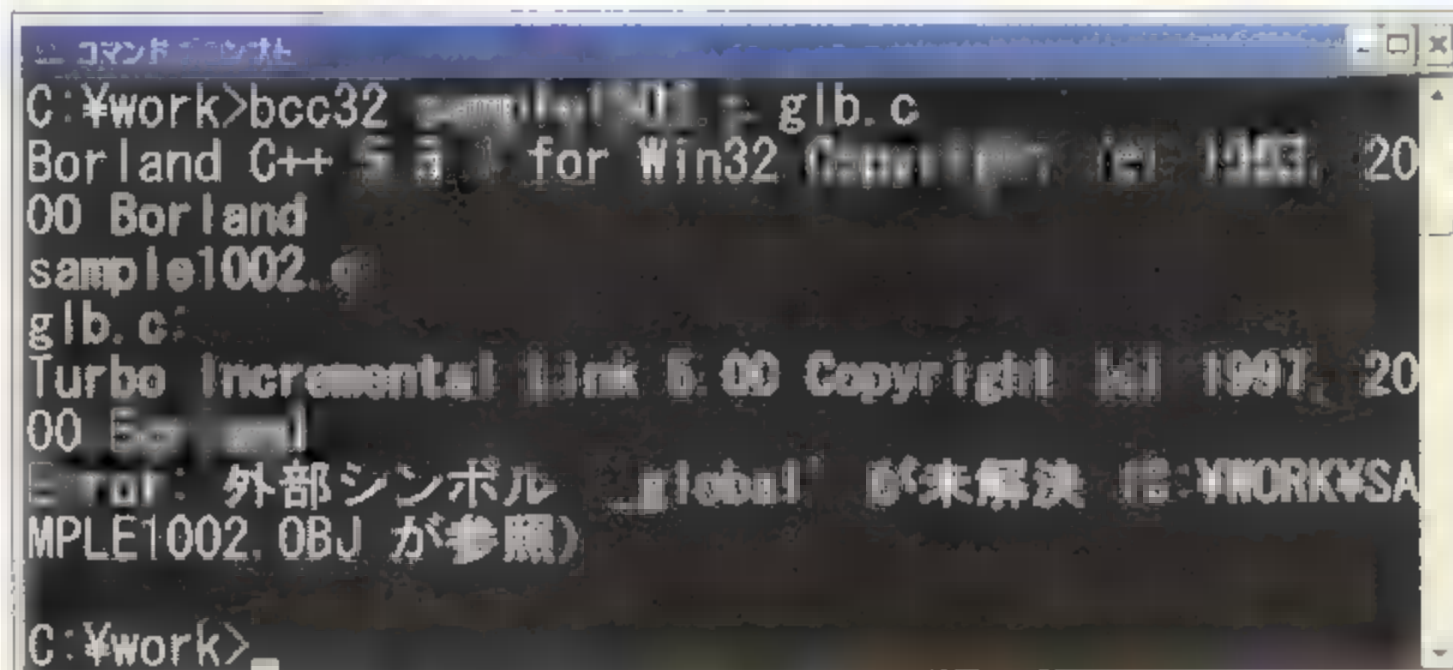
Sample1002.c extern宣言

```
01  #include <stdio.h>
02
03  extern int global;
04
05  /* プロトタイプ宣言 */
06  void dispGlobal(void);
07
08  int main()
09  {
10      global = 5;
11      dispGlobal();
12      global = 10;
13      dispGlobal();
14
15      return 0;
16  }
```

extern宣言を行い、他のソースで宣言したソースを参照します。

extern宣言を行うと、他のソースファイルで宣言した変数にアクセスできます。

```
C:¥work>sample1002
global = 5
global = 10

C:¥work>
```

このように、extern宣言を利用すると、glb.cで宣言したグローバル変数globalにSample1002.Cからアクセスすることができます。

static

externとは逆に、他のソースから利用されないグローバル変数を作成することもできます。この場合、「static」を利用します。P.114のコラムで触れた利用目的とは異なるので注意してください。staticを利用した変数を他のソースファイルからexternで参照しようとした場合、コンパイルエラーが発生します。

構文 static

```
static データ型 変数名;
```

```
C:¥work>bcc32 sample1002.c glb.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
sample1002.c:
glb.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland
Error: 外部シンボル '_global' が未解決（C:¥WORK¥SAMPLE1002.OBJ が参照）

C:¥work>
```

Section 35

定数ラベルとマクロの定義

覚えておきたいキーワード
- 定数
- #define
- マクロ

配列を宣言する場合、要素数は「定数」で指定します。しかし、配列の要素数に直接値を記述すると、要素数を変更する場合に関連するすべてのコードを探す必要があるので面倒です。define文を利用すれば、一からまとめて変更できる定数ラベルを宣言できます。

1. 定数の宣言

■ 定数とは…

「**定数**」とは、プログラムを通じて変更する必要がない数値のことをいいます。たとえば、処理を繰り返す回数などに定数を利用します。

```
for(i=0; i<10; i++) { ……
```

処理を繰り返す回数を表す「10」は、
プログラムを通じて変更する必要がない「定数」です。

では、プログラムを作成した後に「やっぱり繰り返す回数は15回にしてくれ」といわれたら、前述のようなソースコードでは「10」の部分を「15」に書き直す必要があります。**for**文がこの1カ所にしかないならここを修正するだけですが、**for**文が100カ所以上あった場合にはすべてを修正するのはかなり面倒な作業です。

■ 定数ラベルの利用

「**定数ラベル**」を利用すると、たとえばこういった「繰り返し処理の回数」などをあらかじめ1カ所でまとめて宣言することができます。後で回数を変更する場合でも、定数ラベルの宣言部分を修正するだけですむので便利です。

定数ラベルを宣言するには「**#define**」を利用します。これを「**define**文」などと呼びます。**define**文を利用するには、次のように記述します。

構文 define文（定数ラベル）

```
#define 定数ラベル名 定数
```

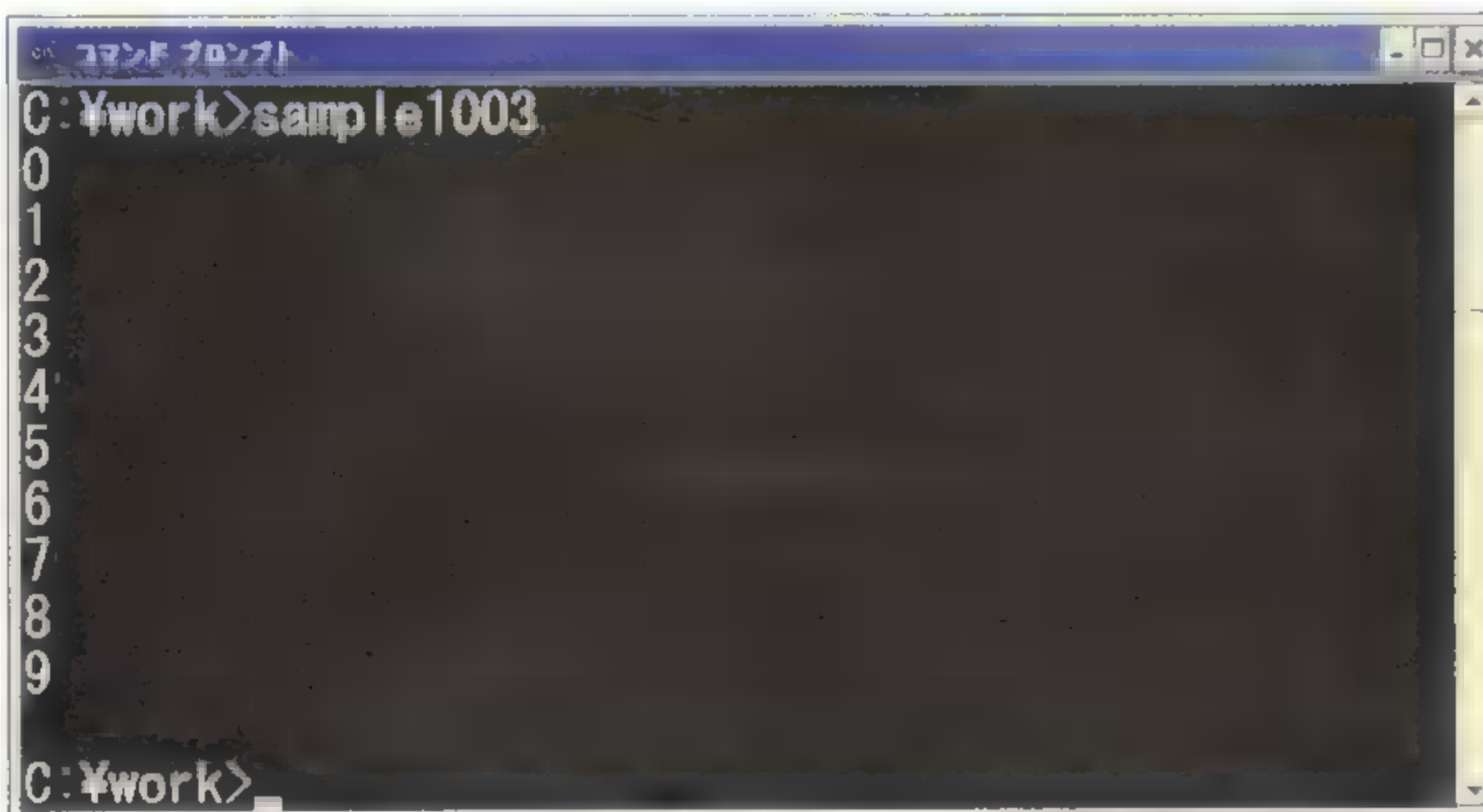
define文はプリプロセッサ命令なので、行末に「**;**（セミコロン）」は付けません。

定数ラベルを利用したプログラムは、次のようになります。

Sample1003.c 定数ラベルの宣言

```
01 #include <stdio.h>
02
03 #define LOOP_NUM 10
04
05 int main()
06 {
07     int i;
08     for(i=0; i<LOOP_NUM; i++) {
09         printf("%d¥n", i);
10     }
11
12     return 0;
13 }
```

定数ラベル「LOOP_NUM」を宣言しています。

数値を直接入力する代わりに定数ラベルを利用します。

■ 配列要素の数を表す定数ラベル

配列要素の数の宣言にも定数ラベルを利用できます。配列要素の数の宣言に変数は利用できませんが、定数ラベルで配列の長さを宣言しておくと便利です。

```
#define STR_LEN   256
  ⋮
char str[STR_LEN];
```

配列の長さに定数ラベルを利用すると、バッファオーバーランなどの単純で深刻なバグを防ぐことができます。

```
for(i=0; i<STR_LEN; i++) {
    str[i] = '¥0';
}
```

STR_LENを変更すると、配列の長さと同時にループの回数も変更されます。

配列の長さを変更したのに、ループ回数の変更を忘れてバッファオーバーランを起こすといったバグを防げます。

2. マクロ

define文を利用して、関数と同様の処理を記述することもできます。これを「**マクロ**」と呼びます。関数ではデータ型のチェックなどが行われますが、マクロでは行われません。マクロは、ソースコード上の指定した部分に、あらかじめ宣言しておいたコードを置き換える機能です。

マクロを宣言するには、次のように記述します。

マクロを利用したプログラムは、次のようになります。

```
#define PLUS(x,y)  x + y
int main()
{
    printf("%d¥n", PLUS(2,3) );
        ⋮
```

「PLUS(2,3)」の部分が、マクロの宣言のとおり「2 + 3」に置き換えられます。

```
C:¥work>
5

C:¥work>
```

PLUS(2,3)の実行結果として「5」と表示されています。

マクロは、データ型のチェックを行わないため、関数の呼び出しに比べてほんのわずかだけ処理時間が高速になります。一般的に、マクロには、関数として定義するまでもない小さな処理を記述する場合がほとんどです。データ型のチェックが行われないことから、バグが混入する可能性が高くなるので、マクロでは複雑な処理は行わないように注意します。

Column const修飾子

defineによる定数ラベルの宣言と似た機能に「const修飾子」があります。const修飾子を利用すると、「値の変えられない変数」を宣言することができます。

これは、うっかりミスで値を変更してしまわないための変数として用いることができます。また、値を変更できないので、宣言と同時に初期値を設定する必要があります。

構文 const修飾子

```
const データ型 変数名 = 定数;
```

たとえば、int型の定数変数を宣言するには、次のように記述します。

```
const int str_len = 10;
```

また、関数の引数などでも利用できます。

```
void foo(const int count)
    ⋮
```

値を変更されない引数を宣言できます。

Section 36

条件付きコンパイル

覚えておきたいキーワード

- #if～#endif
- #if～#else～#endif
- #ifdef、#ifndef

プリプロセッサ命令によって、ソースコードの一部をコンパイルしないように記述することができます。条件に応じてコンパイルしたり、しなかったりを設定できるので、これらのプリプロセッサ命令のことを「条件付きコンパイル」と呼びます。

1. 部分的なコンパイル

#if～#endif

C言語では、ソースコードの一部を条件によってコンパイルしないようにすることができます。これを「**条件付きコンパイル**」と呼びます。

条件付きコンパイルを行うには「**#if**」と「**#endif**」を利用します。

構文 #if～#endif

```
#if 条件式
……ソースコード……
#endif
```

「**#if**」に続けてスペース文字やタブで区切って条件式を入力します。条件式の結果が「真(0以外)」の場合はソースコードがコンパイルされ、「偽(0)」の場合はコンパイルされません。また、条件に変数は利用できないため、定数ラベルなどを利用した条件式を指定します。

条件付きコンパイルを行いたいソースコードの範囲の終端には、「**#endif**」を記述します。

条件付きコンパイルを利用したプログラムは、次のようになります。

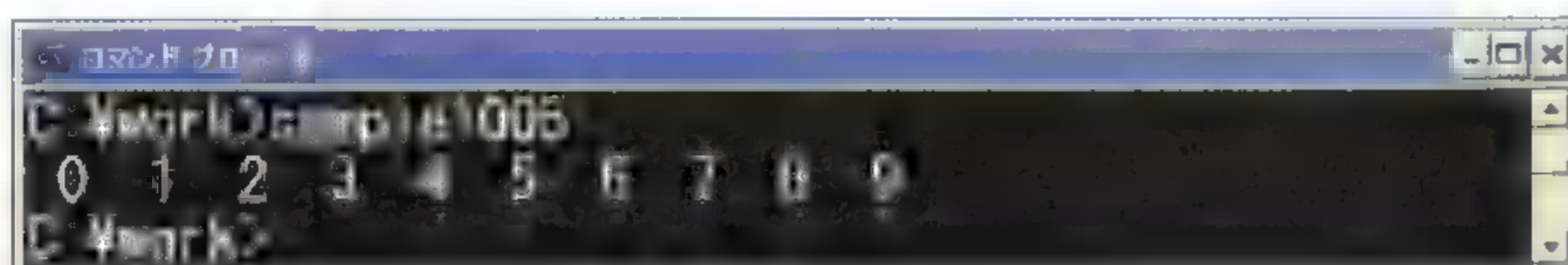
Sample1005.c 条件付きコンパイル

```
01  #include <stdio.h>
02
03  #define VAL 10
04
05  int main()
```

```
06  {
07      int i;
08  #if (VAL == 10)
09      for(i=0; i<VAL; i++) {
10          printf("%2d ", i);
11      }
12  #endif
13      return 0;
14  }
```

定数ラベルVALが10であれば、続きのコードをコンパイルします。

条件付きコンパイルを行う範囲の終端を示します。

```
C:\work>sample1005
 0  1  2  3  4  5  6  7  8  9
C:\work>
```

この例では、**#if**に続く条件式が真なのでコンパイルが行われます。プログラムを実行すると9～11行目が処理され、数値が表示されます。

3行目の定数ラベルの宣言を次のように変更すると、**#if**に続く条件式が偽になり、9～11行目のコンパイルが行われません。

```
#define VAL 9
```

このように変更したプログラムを実行すると、9～11行目の処理が行われないため何も表示されません。

■ #if～#else～#endif

if文では条件式を満たさない場合、**else**文の後に記述した処理が実行されます。それと同じように、**#if**にも「**#else**」があります。

構文 #if～#else～#endif

```
#if 条件式
    ……条件式が真のときにコンパイルされるコード……
#else
    ……条件式が偽のときにコンパイルされるコード……
#endif
```

また、「**else if**」のように続けて条件判断を行いたい場合には「**#elif**」を利用します。

構文 #if～#elif～#endif

```
#if 条件式1
    ……条件式1が真のときにコンパイルされるコード……
#elif 条件式2
    ……条件式1が偽で、条件式2が真のときにコンパイルされるコード……
#else
    ……条件式1、条件式2が偽のときにコンパイルされるコード……
#endif
```

#ifdef、#ifndef

#ifでは、その後に記述した条件式を評価してその結果を判断します。それとは別に「ある定数ラベル名もしくはマクロ名が宣言されていたら」という条件で条件付きコンパイルを行うこともできます。これには「**#ifdef**」および「**#ifndef**」を利用します。

構文 #ifdef

```
#ifdef 定数ラベル名またはマクロ名
    ……宣言されている場合にコンパイルされるコード……
#endif
```

構文 #ifndef

```
#ifndef 定数ラベル名またはマクロ名
    ……宣言されていない場合にコンパイルされるコード……
#endif
```

ifdef文や**ifndef**文にも**#else**や**#elif**を利用することもできます。

2. ヘッダーファイルの重複

■ ヘッダーファイル内でのinclude文

プログラムの規模が大きくなると、あるヘッダーファイルが、さらに他のヘッダーファイルを**include**文により取り込んでいる場合があります。この場合、知らず知らずのうちに同じヘッダーファイルを重複して取り込んでしまうことがあります。

たとえば、次のような場合が考えられます。

human.h human構造体の定義

```
01  typedef struct human {
02      char name[32];
03      int age;
04  } HUMAN;
```

school.h school構造体の定義

```
01  #include "human.h"
02  typedef struct school {
03      HUMAN teacher[10];
04      HUMAN student[256];
05  } SCHOOL;
```

この2つのヘッダーファイルがある場合に、次のようなソースコードを記述すると、同じヘッダーファイルを2回取り込むことになります。

Sample1005.c

```
01  #include "human.h"
02  #include "school.h"
03  ⋮
```

これらのファイルを用意してコンパイルすると、次のようなエラーが発生します。

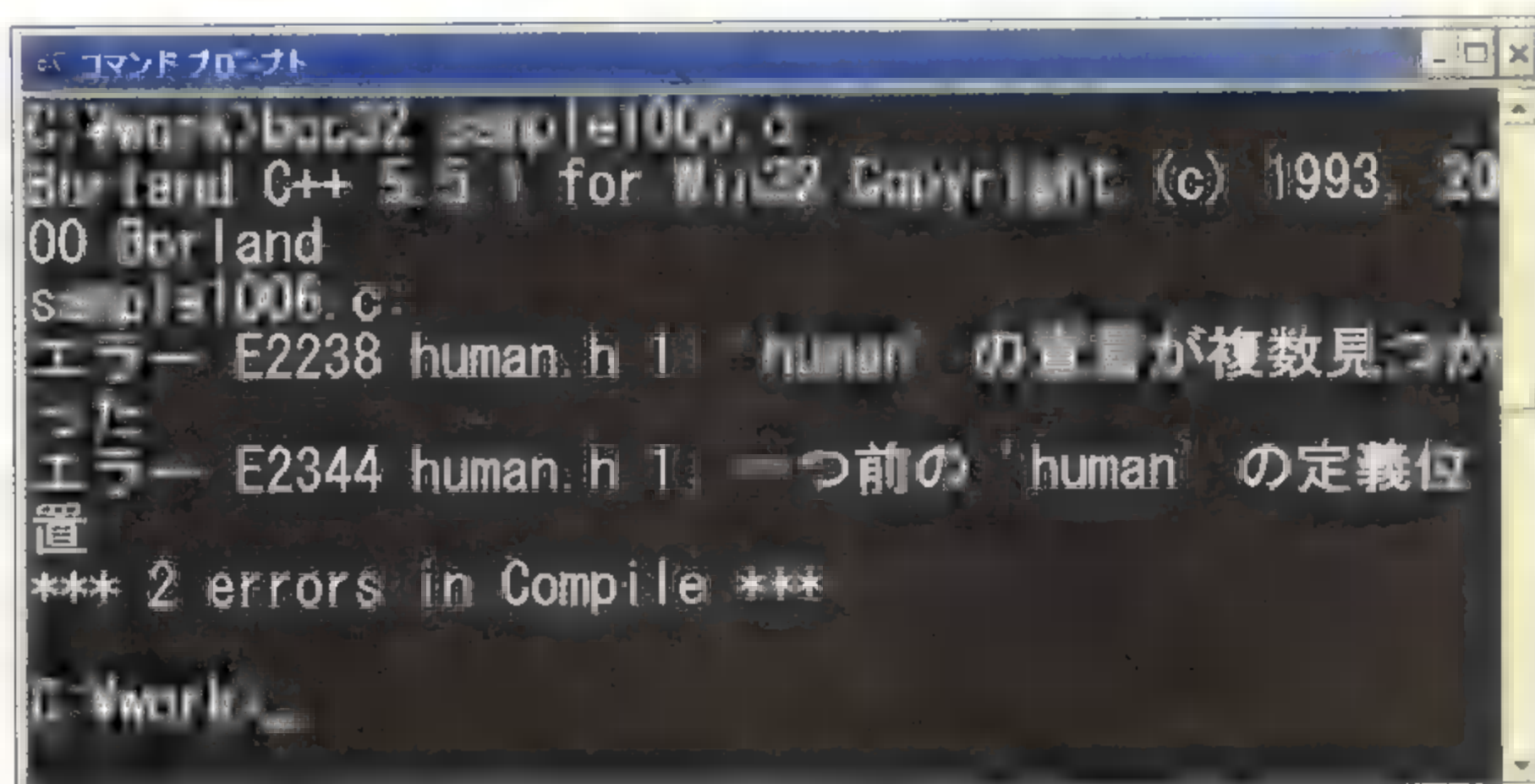
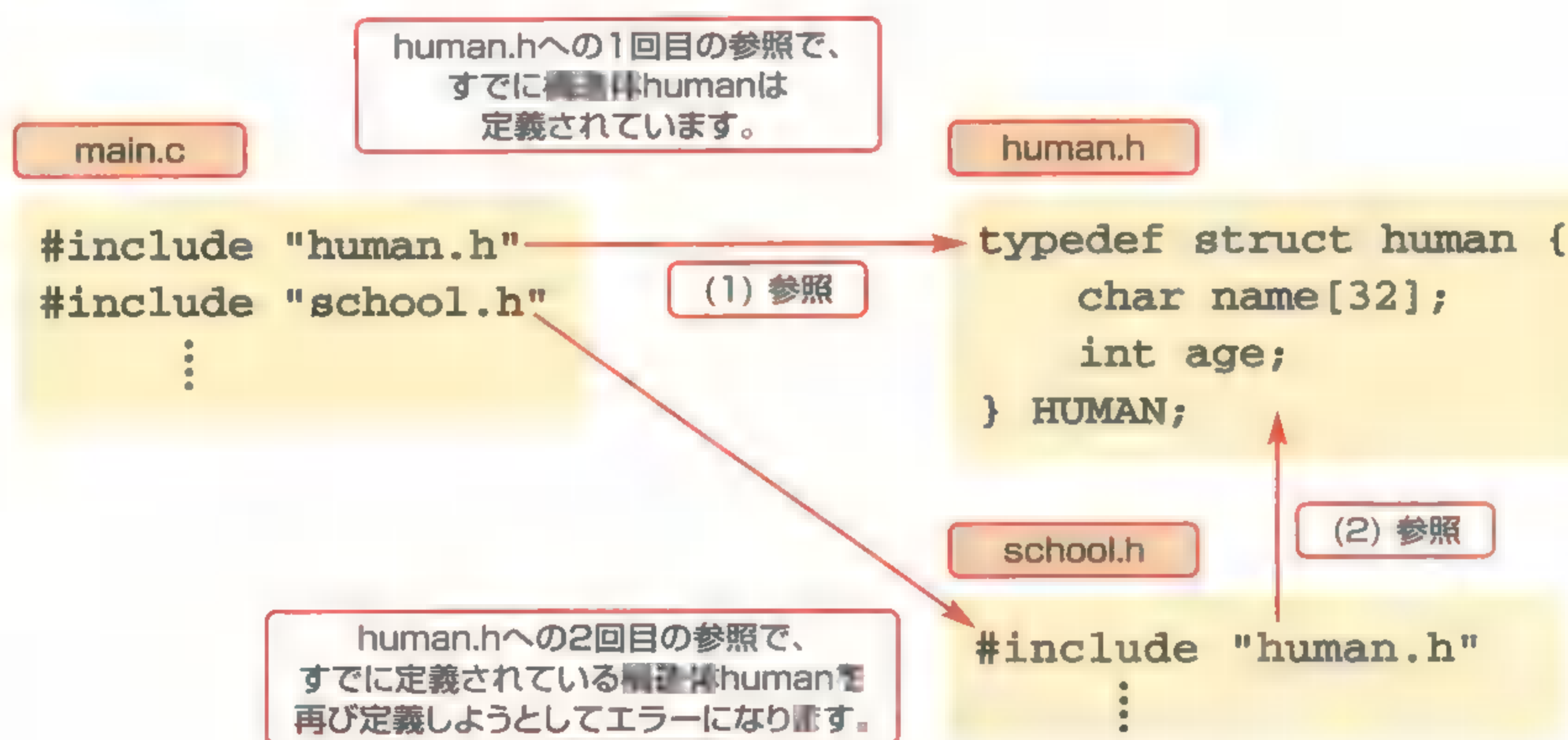
```
C:¥work>bcc32 sample1006.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
sample1006.c:
エラー E2238 human.h 1: 'human' の宣言が複数見つかった
エラー E2344 human.h 1: 一つ前の 'human' の定義位置
*** 2 errors in Compile ***

C:¥work>_
```

これは、Sample1006.cからhuman.hが読み込まれ、次にschool.hを読み込んだ際にschool.hの先頭で再びhuman.hが読み込まれており、その結果としてSample1006.cから見ると**human**構造体が2回目の定義に見えるため、コンパイルエラーが発生します。

図1 ヘッダーの重複

この例では、単純にSample1006.cにおいてhuman.hを取り込む**include**文を記述しないようにすれば解決しますが、プログラムの規模が大きくなると簡単に重複を避けられない場合があります。

この問題を回避するために、条件付きコンパイルを利用します。

■ 重複したファイルのコンパイルの禁止

条件付きコンパイルを利用して、一度**include**文で取り込んだヘッダーファイルはもうコンパイルしないようにします。このような指定を行うには**define**文と**ifndef**文を組み合わせて利用します。

human.h　2回以上読み込まれないヘッダー

```
01  #ifndef HEADER_HUMAN_H
02  #define HEADER_HUMAN_H
03  typedef struct human {
04      char name[32];
05      int age;
06  } HUMAN;
07  #endif
```

ここでHEADER_HUMAN_Hを定義します。

2回目からは、HEADER_HUMAN_Hが定義されているため、このヘッダーはコンパイルされません。

このように記述することで、構造体の定義を1回目だけコンパイルすることができ、エラーを回避することができます。

Column　#undef

「**#undef**」を利用すると、**#define**を用いて宣言した定数ラベルやマクロを、未宣言の状態に戻すことができます。条件付きコンパイルでは、**#ifdef**などで「定数ラベルなどが宣言されているか、いないか」を条件とするので、一度宣言した定数ラベルなどを未宣言の状態に戻す方法が必要となることがあります。
また、一度宣言した定数ラベルなどを違う値で宣言し直したい場合などにも**#undef**を利用します。**#undef**を利用するには、次のように記述します。

構文　#undef

```
#undef　定数ラベル名またはマクロ名
```

#undefを利用したプログラムは、次のようになります。

```
#ifdef HEADER_HUMAN_H
#undef HEADER_HUMAN_H
#define HEADER_HUMAN_H 1
#endif
```

HEADER_HUMAN_Hを未宣言の状態にします。

HEADER_HUMAN_Hを、値「1」の定数ラベルとして宣言し直します。

まとめ　第10章：プリプロセッサ命令

この章では、コンパイルの準備のための処理を記述するプリプロセッサ命令について学習しました。プリプロセッサ命令は、コンパイラがC言語の構文をチェックするよりも前にあらかじめ処理されるため、定数ラベルを定義したり、部分的にコンパイルを行わないようにしたりできることなどを学びました。

第10章で学習したこと

- プリプロセッサ命令とは、コンパイルを行う前に、その準備のためにあらかじめ処理される記述である。
- #includeは、ヘッダーファイルを取り込むときに利用する。
- ヘッダーファイルには関数のプロトタイプ宣言や、構造体の定義などを記述し、複数のソースファイルから利用することができる。
- #defineを利用して、定数ラベルを宣言することができる。
- #defineでは、関数のようなマクロを宣言することができる。ただしマクロは引数の型のチェックを行えない。
- 条件付きコンパイルを利用すると、ソースコードを部分的にコンパイルしたり、しなかったりを設定できる。
- ヘッダーファイルを重複して取り込んでしまう場合は、#defineと#ifdefを利用して2回目以降はコンパイルを行わないようにようにする。

ステップアップ！

プリプロセッサ命令は、#includeと#define以外は、入門レベルのプログラミングでは利用頻度が低いかもしれません。逆に、#includeと#defineについてはきっちりと覚えておきましょう。特に#defineを使った定数ラベルは、非常に便利で重要です。

問1 include文とヘッダーファイル

次の関数のプロトタイプ宣言を行うヘッダーファイルを作成し、ソースファイルからそのヘッダーファイルを取り込んでください。

```
void disp(void)
{
    printf("こんにちは¥n");
}
```

答1

ヘッダーファイルを「disp.h」という名前で保存した場合の解答例を示します。

```
void disp(void)
{
    printf("こんにちは¥n");
}
```

ソースファイルから、このヘッダーファイルを**include**文で取り込む場合、ファイル名は「"(ダブルクォーテーション)」で囲みます。

```
#include "disp.h"
```

問2 定数ラベルの宣言

定数ラベルを利用して、ループ処理の回数を記述しました。次のソースコードの空欄部分を埋めてください。

```
#        LOOP_NUM    10

void loop(void)
{
    int i;
    for(i=0; i<        ; i++) {
        ⋮
```

答2

定数ラベルを利用するには、**#define**を用います。また、定数ラベルは数値として扱うことができます。

```
#define    LOOP_NUM   10

void loop(void)
{
    int i;
    for(i=0; i<LOOP_NUM; i++) {
        ⋮
```

問3 2回目にはコンパイルしないヘッダーファイル

重複して取り込まれないようなヘッダーファイルを作成しました。次のソースコードの空欄部分を埋めてください。

```
#        MYHEADER_H
#        MYHEADER_H
```

答3

重複して取り込まれないようにするためには、**#ifndef**と**#define**を利用します。

```
#ifndef    MYHEADER_H
#define    MYHEADER_H
```

付録

Visual Learning Introduction of C

開発環境の設定

付録
1

覚えておきたいキーワード
- 拡張子の表示や変更
- 環境変数
- PATH

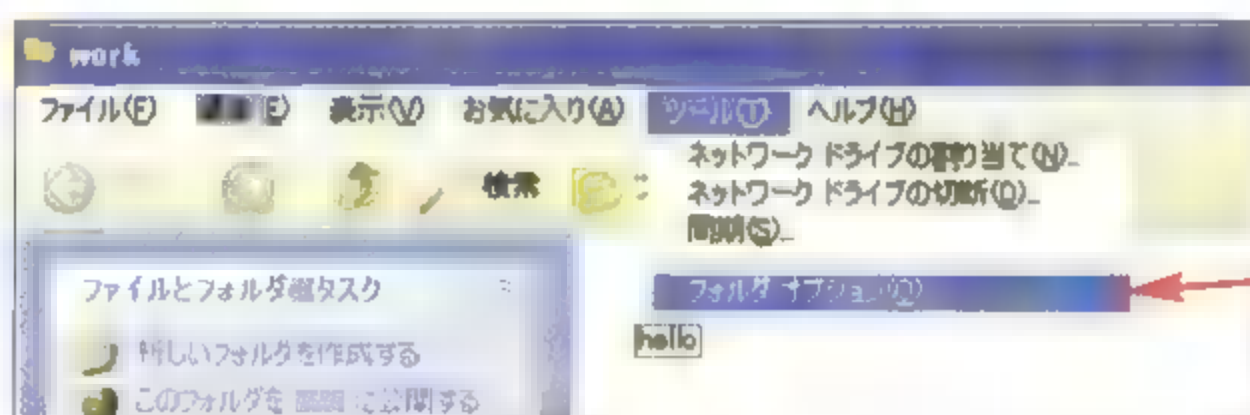
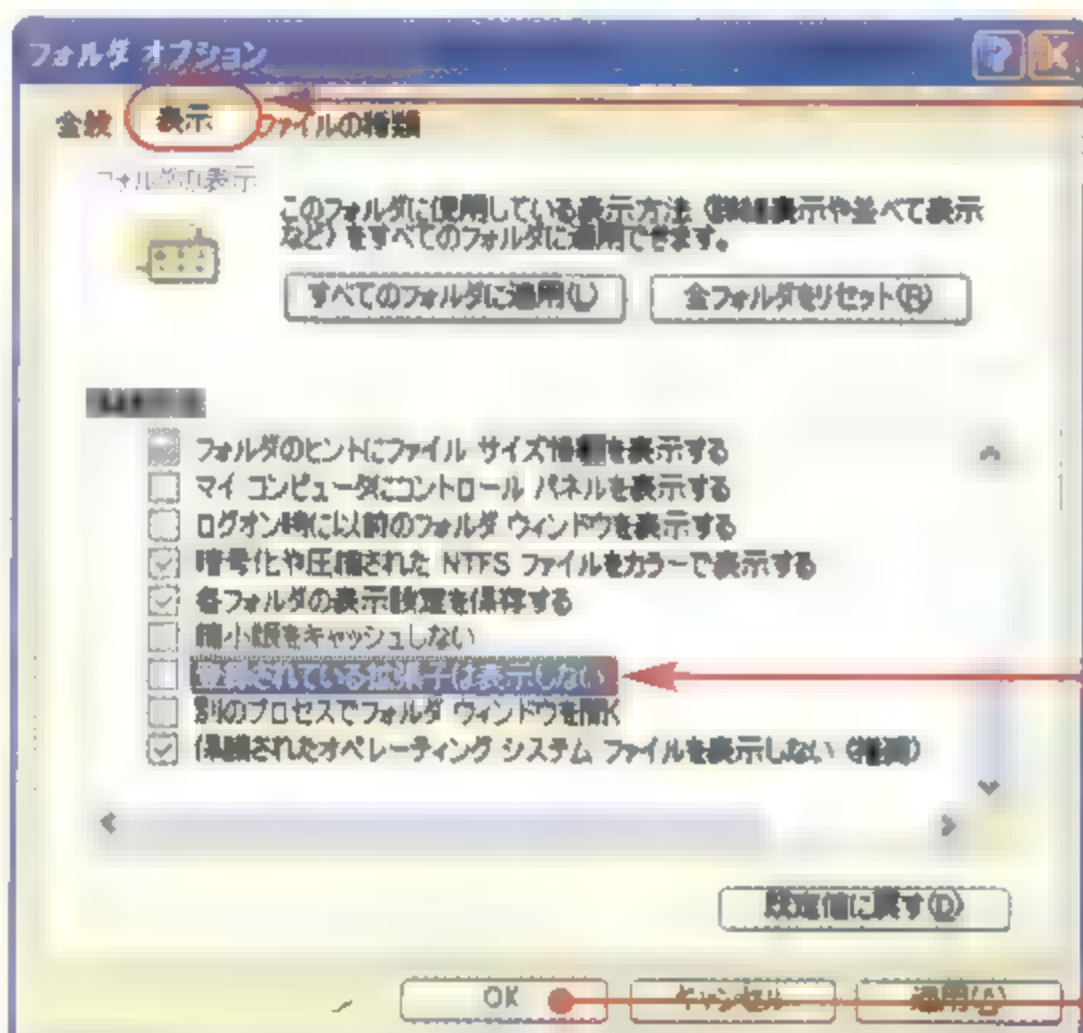
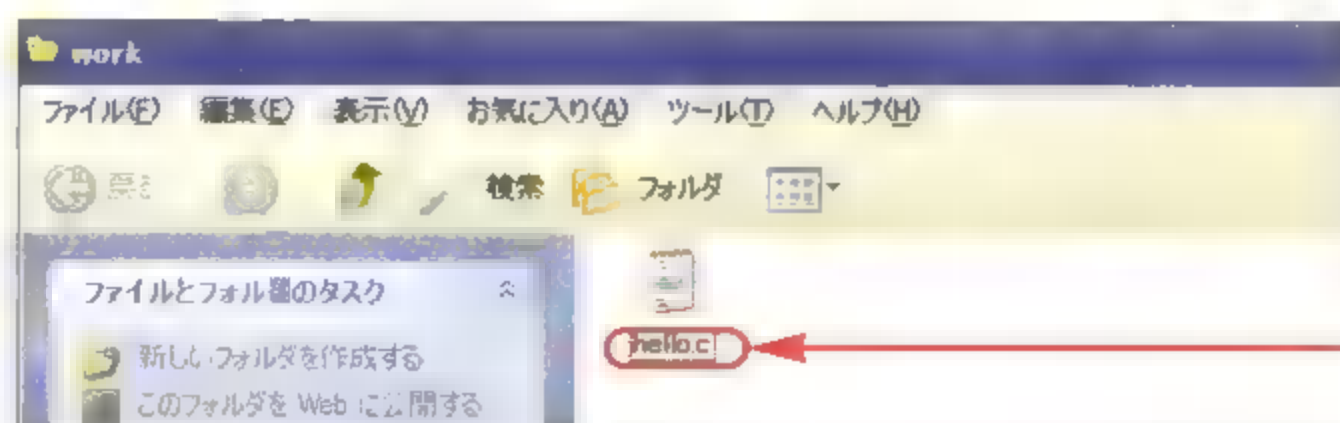
コンパイラのインストール

本書ではBorland C++ Compiler 5.5を利用しています。ここではそのコンパイラのダウンロードからインストール、利用開始までに必要な準備について、手順を追って解説しています。コンパイラの入手法や、設定方法などがわからないときの参考にしてください。

1. ファイルの拡張子の表示

拡張子が表示されていない場合は…

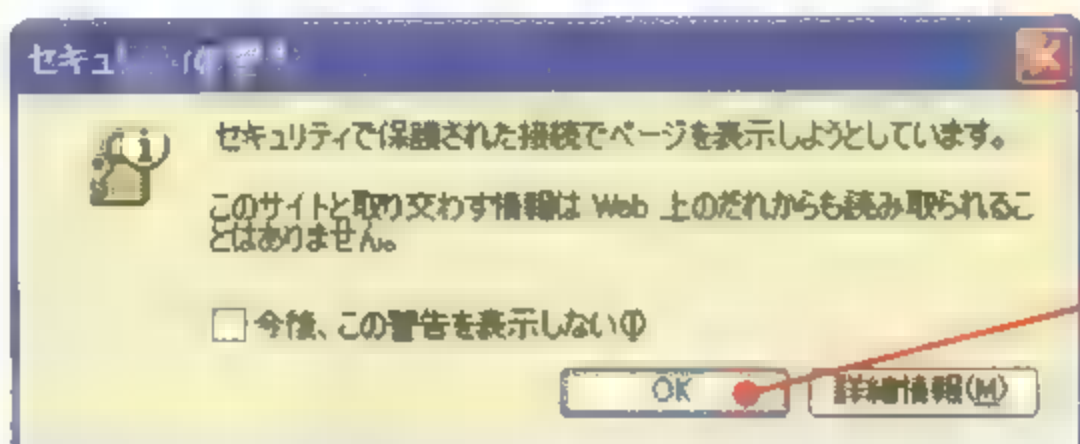
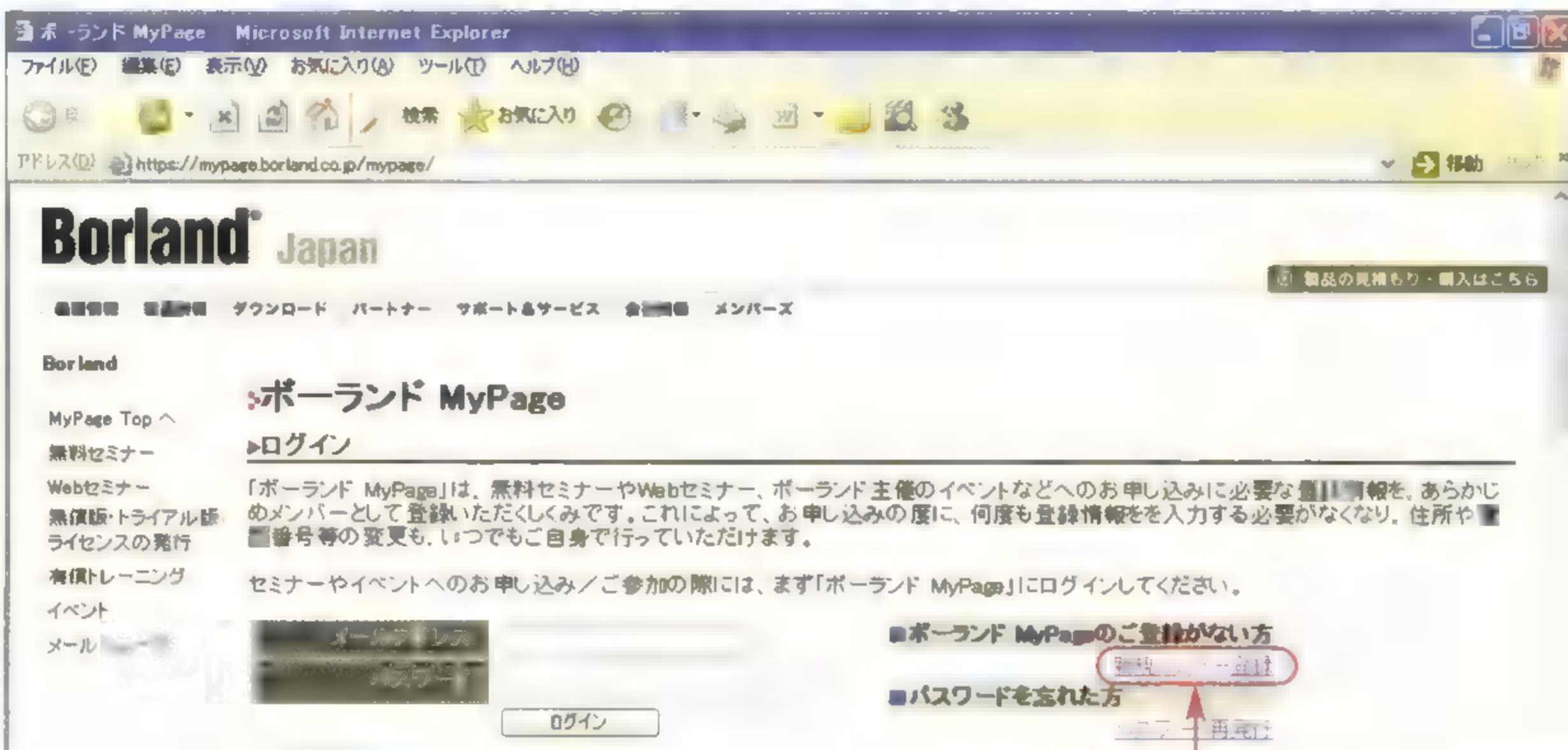
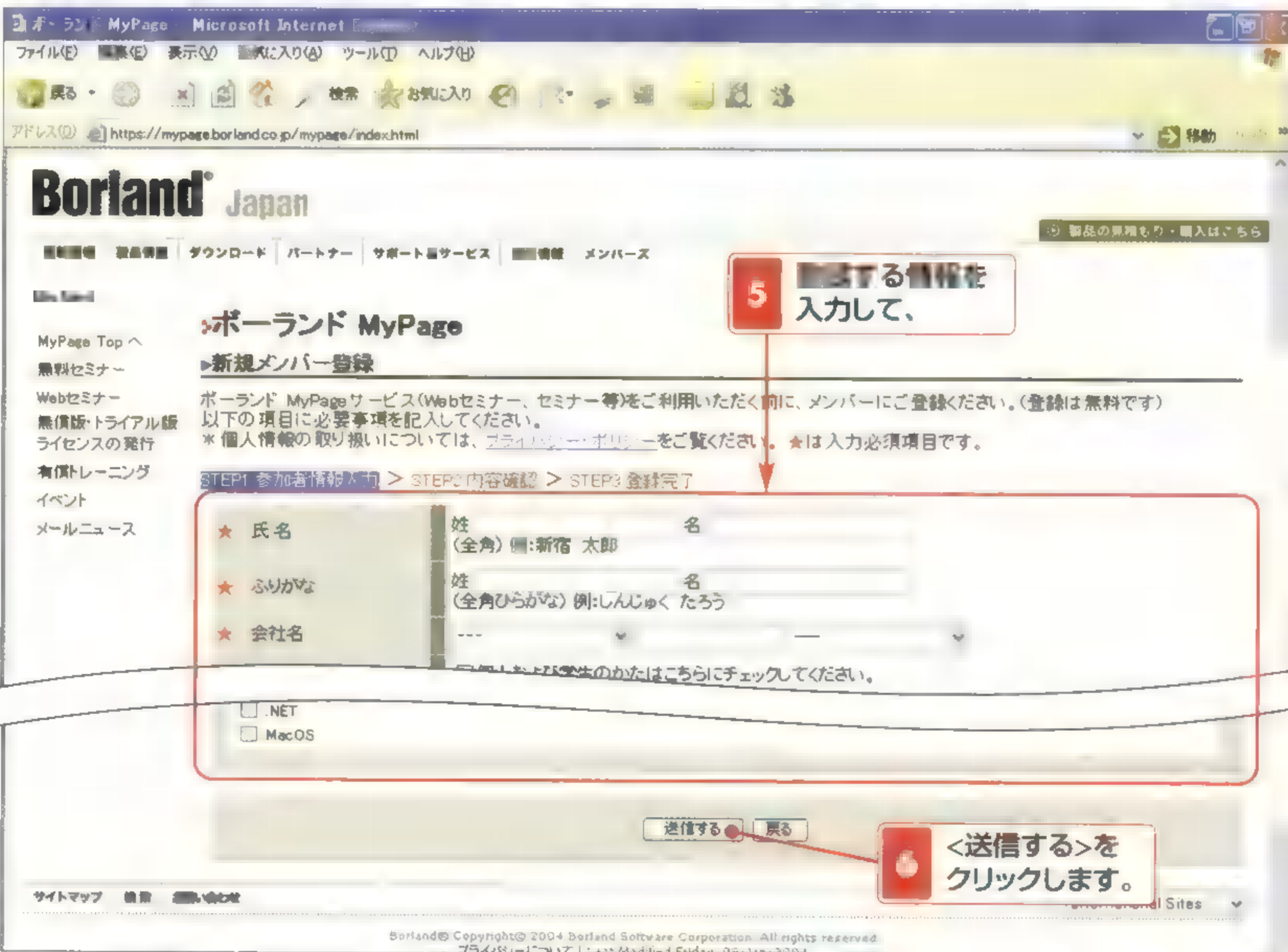
Windowsは標準で「登録されている拡張子は表示しない」という設定になっています。ファイルの拡張子が表示されないと、作業を行う上で不便な場合がありますので、拡張子が常に表示されるように変更します。

設定を変更するには、以下の手順に従います。

2. Borland MyPageへの登録

BorlandのWebページから、コンパイラをダウンロードします。そのためには、Borland MyPageにユーザー登録をする必要があります。まず、Borland C++ Compiler 5.5のWebページを表示します。

Borland C++ Compiler 5.5のWebページ

http://www.borland.co.jp/cppbuilder/freecompiler/

以降は、次の手順に従います。

3 <セキュリティの警告>が表示された場合、<OK>をクリックします。

4 <新規メンバー登録>をクリックし、

5 登録する情報を入力して、

ボーランド MyPage

新規メンバー登録

ボーランド MyPageサービス(Webセミナー、セミナー等)をご利用いただく前に、メンバーにご登録ください。(登録は無料です)
以下の項目に必要事項を記入してください。
＊個人情報の取り扱いについては、プライバシーポリシーをご覧ください。★は入力必須項目です。

STEP1 参加者情報入力 > STEP2 内容確認 > STEP3 登録完了

★ 氏名　姓　名　(全角) 例:新宿 太郎
★ ふりがな　姓　名　(全角ひらがな) 例:しんじゅく たろう
★ 会社名

.NET
MacOS

送信する　戻る

6 <送信する>をクリックします。

Borland® Copyright© 2004 Borland Software Corporation. All rights reserved.

ボーランド MyPage - Microsoft Internet Explorer
https://mypage.borland.co.jp/mypage/index.html
Borland Japan
ダウンロード　パートナー　サポート＆サービス　メンバーズ
ボーランド MyPage
新規メンバー登録
STEP1 参加者情報入力 > STEP2 内容確認 > STEP3 登録完了
★ 氏名
★ ふりがな
★ 会社名
送信する　戻る

7 内容を確認して、

8 <送信する>をクリックすると、

9 新規メンバー登録が完了します。

付録

3. コンパイラのダウンロード

新規メンバー登録が完了したら、MyPageにログインしてコンパイラのインストーラをダウンロードします。

Borland C++ Compiler 5.5のWebページを表示します。

Borland
ログイン
メールアドレス
パスワード
ログイン
4 登録したメールアドレスとパスワードを入力して、
5 <ログイン>をクリックします。

Borland Japan
6 MyPageへのログインができるので、
ダウンロードしたい／ライセンスを取得したい製品を選択してください。
C++ Compiler
JBuilder 2005 Enterpriseトライアル
JBuilder 2005 Foundation
JBuilder X Developer plus UML Trial（ライセンスキーのみ）
JBuilder X Foundation（ライセンスキーのみ）
StarTeam 2005 トライアル
Turbo Debugger
7 <C++ Compiler>を選択し、
8 <次へ>をクリックします。
次へ
戻る

製品名：C++ Compiler
C++Compilerのライセンスは、インストールイメージに含まれます。以下のライセンス発行手続きが完了すると、電子メールでダウンロードURLをお送りします。電子メールに記載された手順に従って、インストールを実行してください。
9 <発行する>をクリックすると、
発行する
戻る

Borland

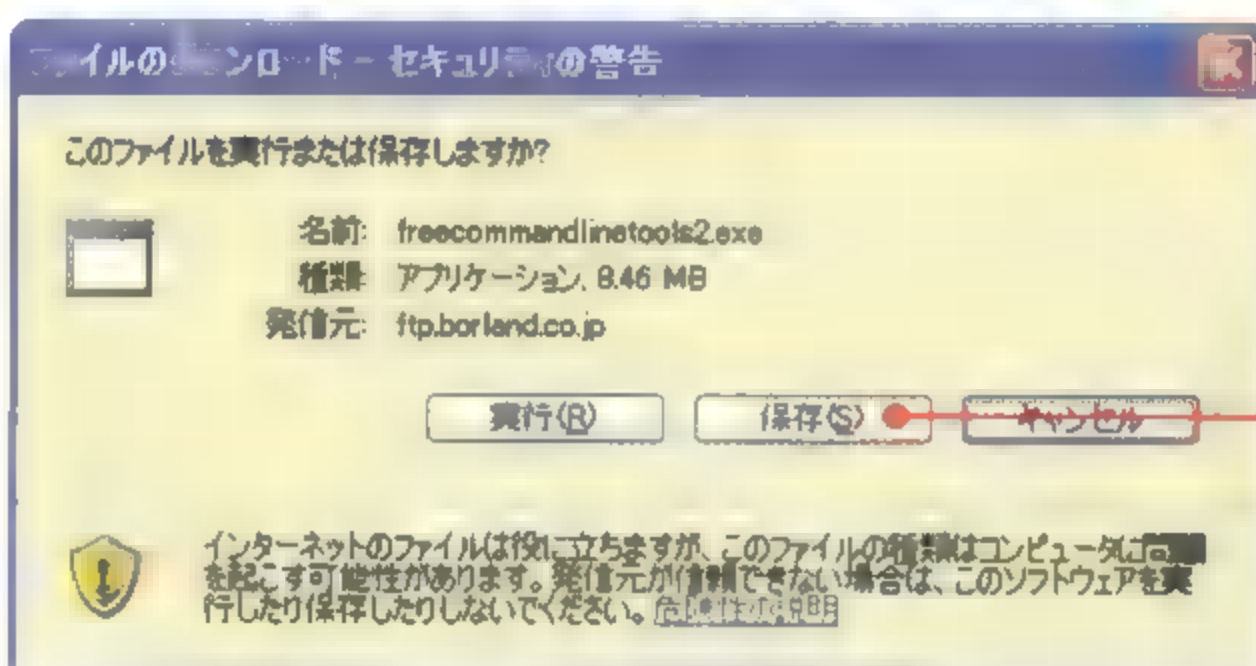
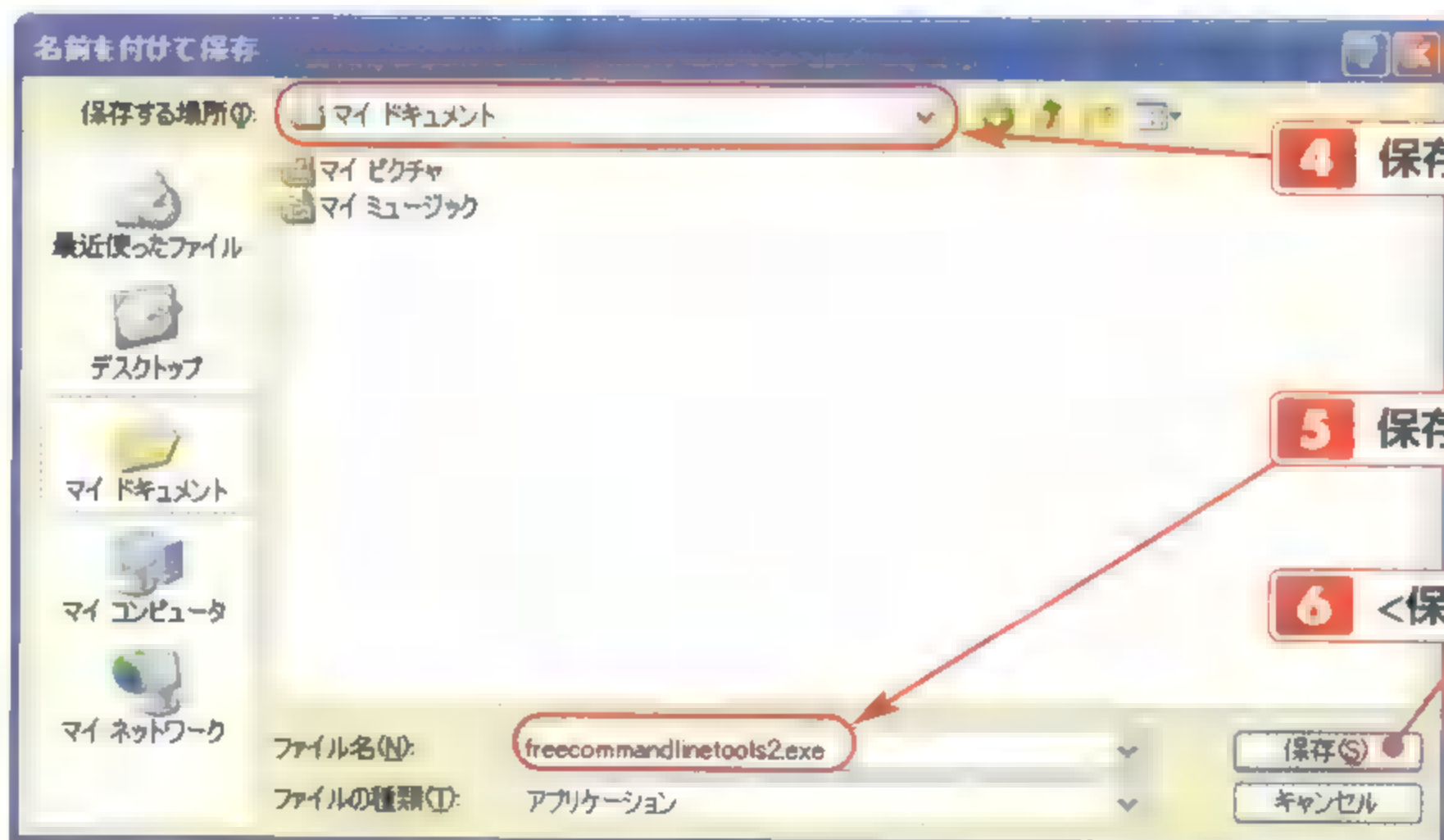
無償版・トライアル版製品ダウンロードおよびライセンスの発行

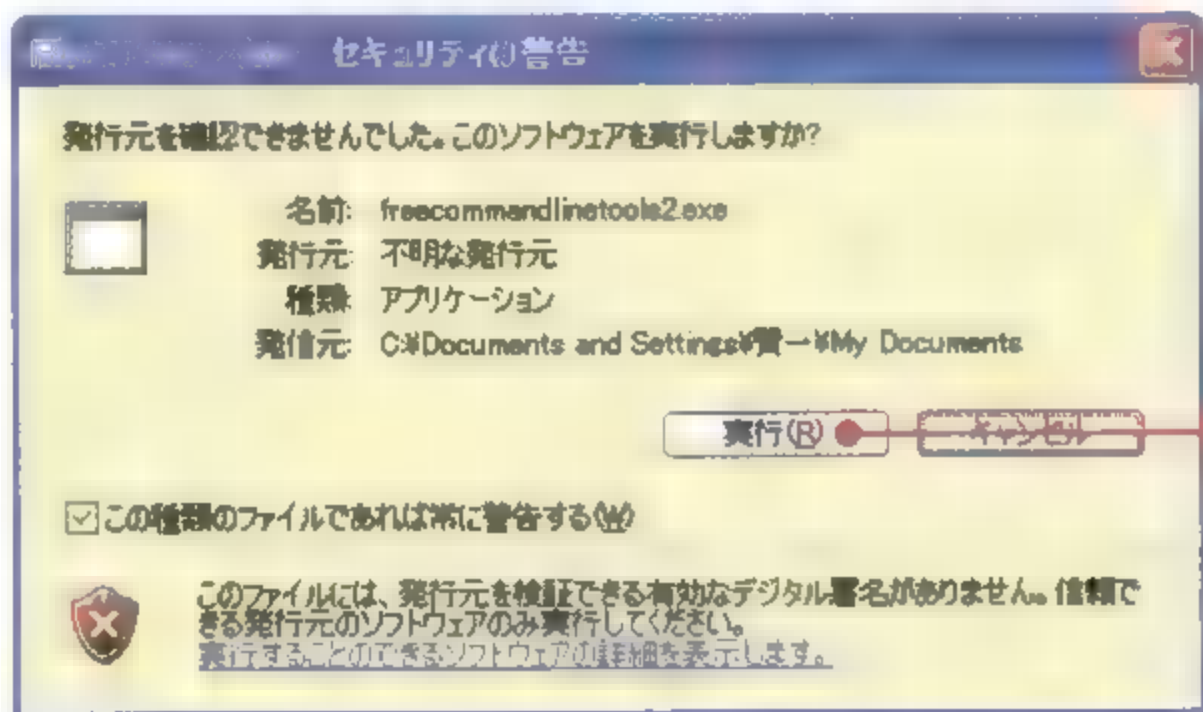
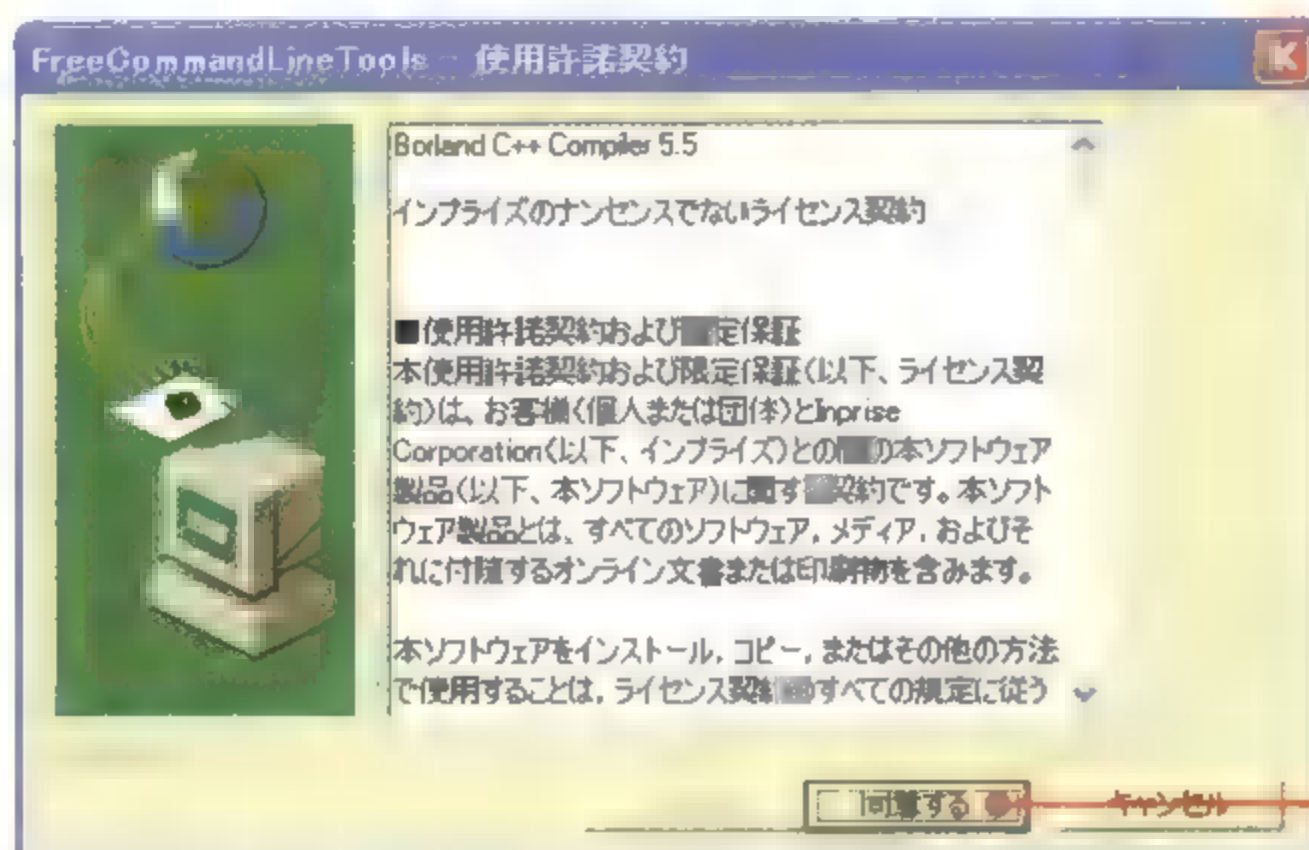
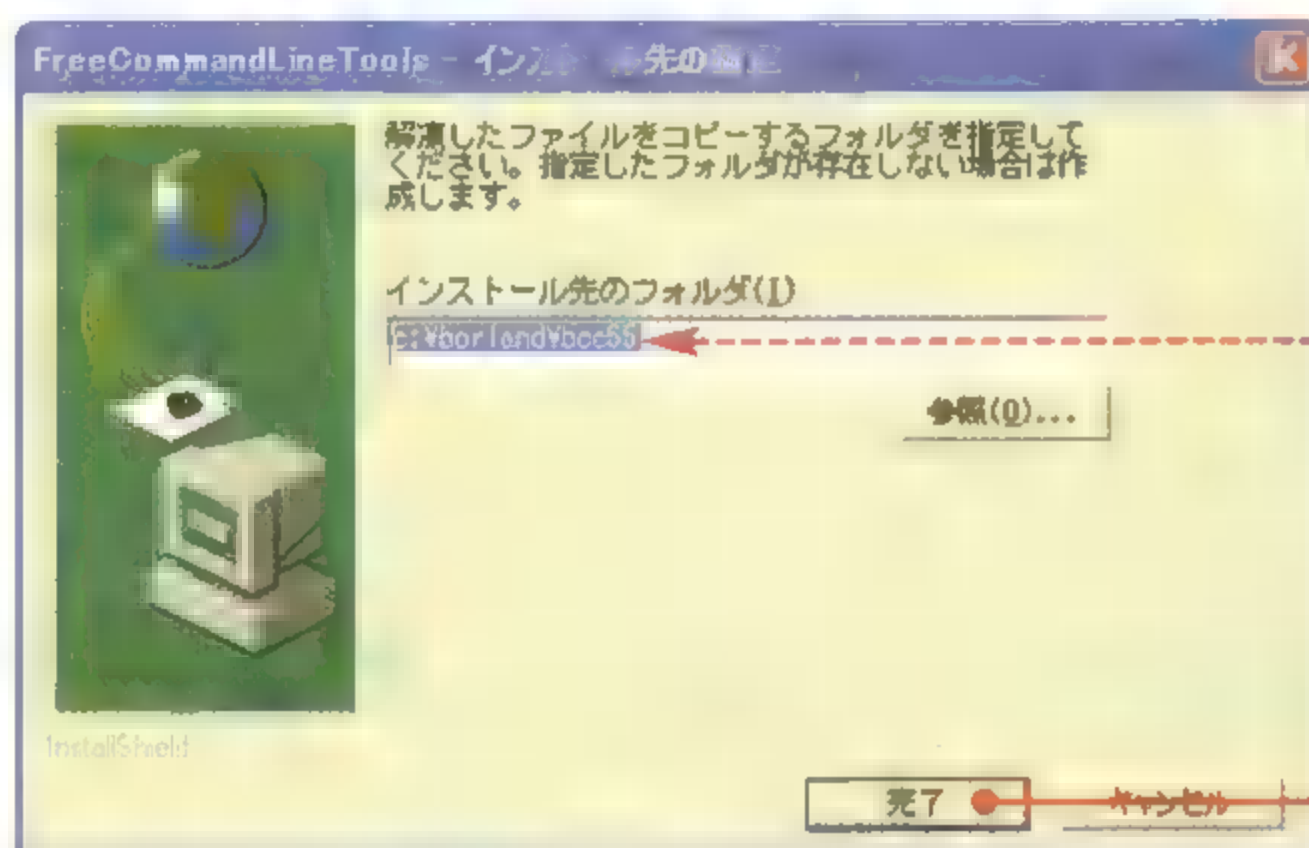
ご登録されているメールアドレス【　】にライセンス情報を送信しました。

10 登録したメールアドレス宛てに、コンパイラをダウンロードできるURLが送信されます。

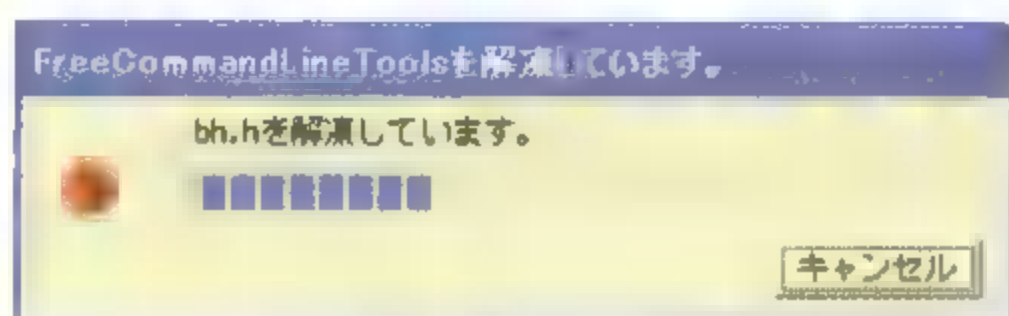
登録したメールアドレス宛てにメールが届いたら、コンパイラのインストーラがダウンロードできるURLが記載されていますので、そのURLをブラウザで開きます。

以降は、次の手順に従います。

メールで指示されたURLをブラウザで開きます。

4. コンパイラのインストール

　ダウンロードが完了したら、次はコンパイラをインストールします。コンパイラのインストーラを保存したフォルダを開いて、以下の手順に従います。

4 <同意する>をクリックし、

コンパイラのインストール先を変更することもできます。

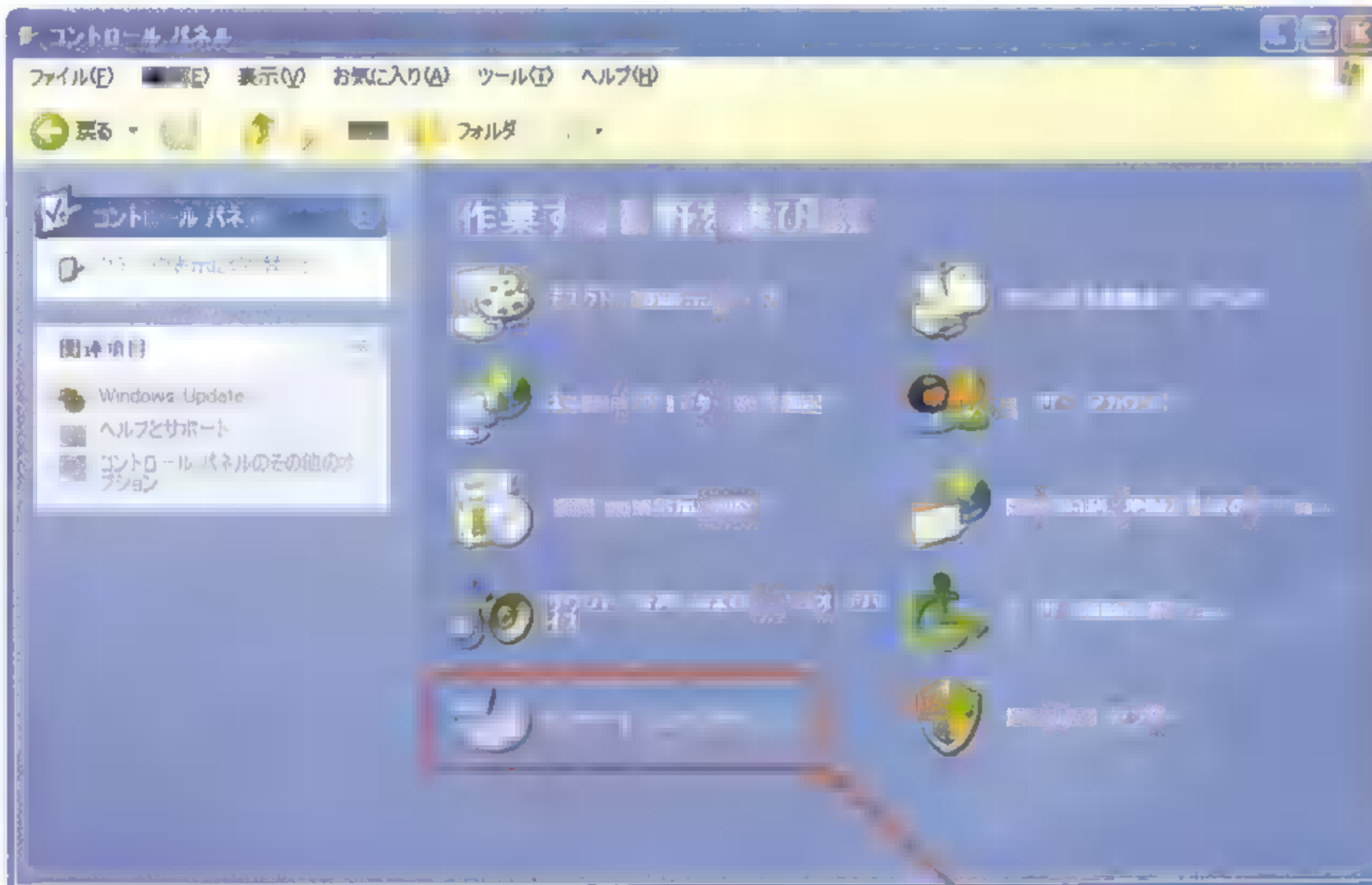
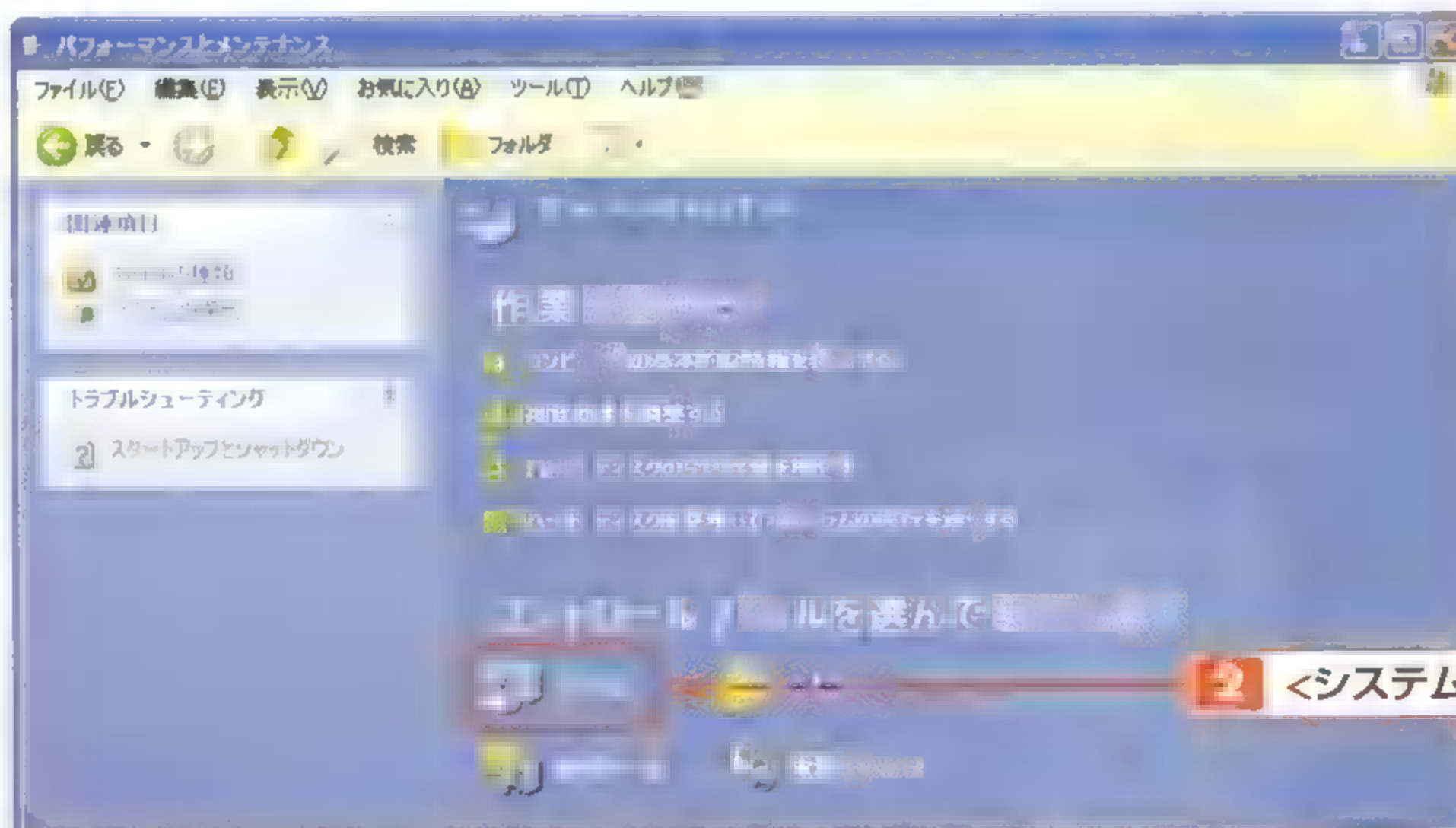
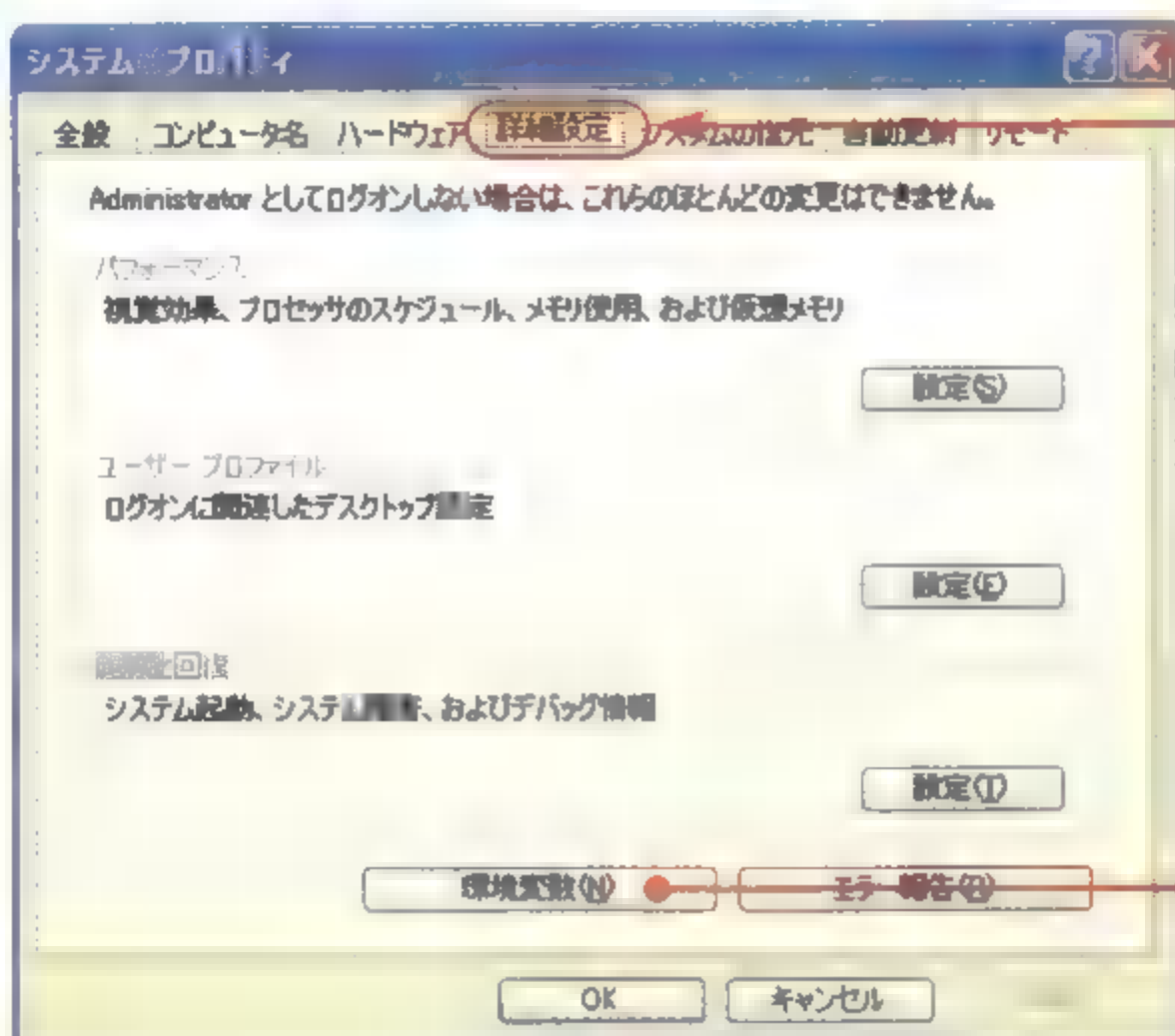
5 <完了>をクリックして、

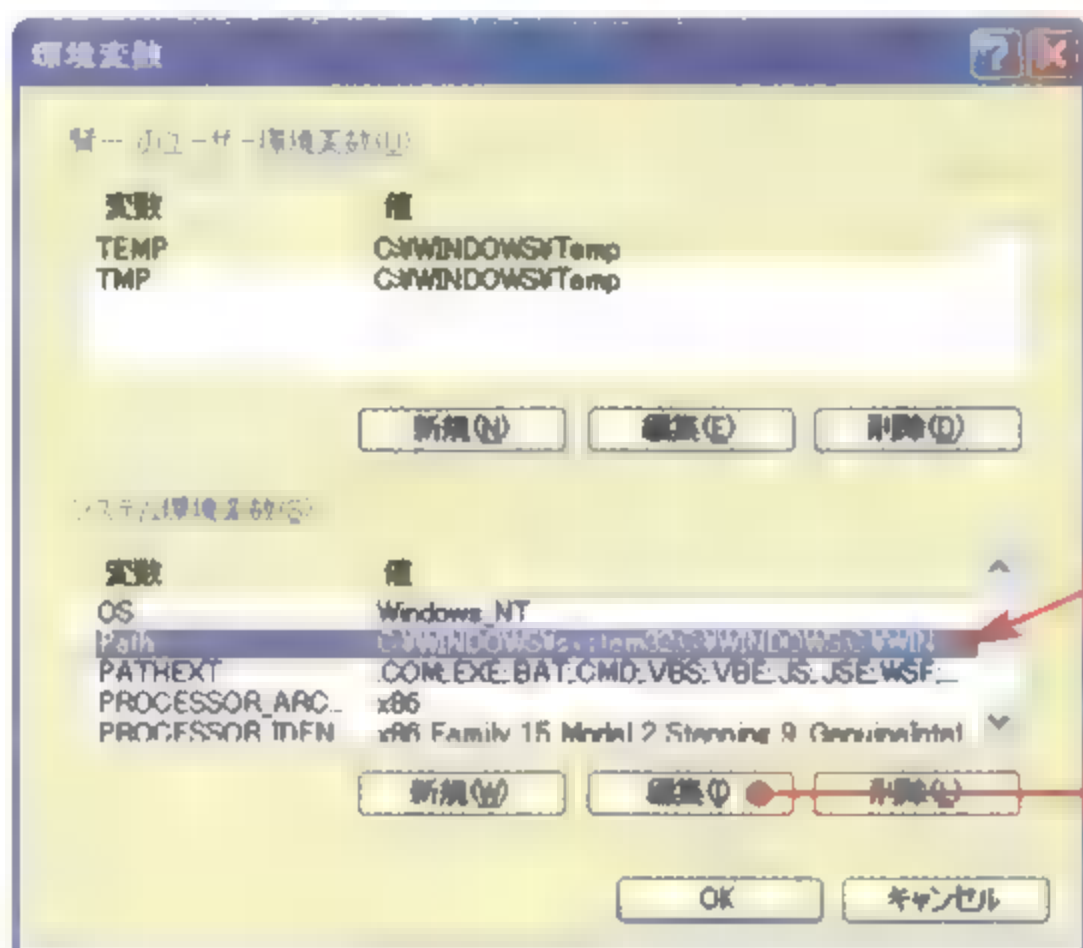
6 インストール先フォルダを新規作成するか確認されるので、<はい>をクリックすると、

7 インストールが開始します。

インストールが終了すると、「パッケージの転送に成功しました。」と表示されます。

8 <OK>をクリックします。

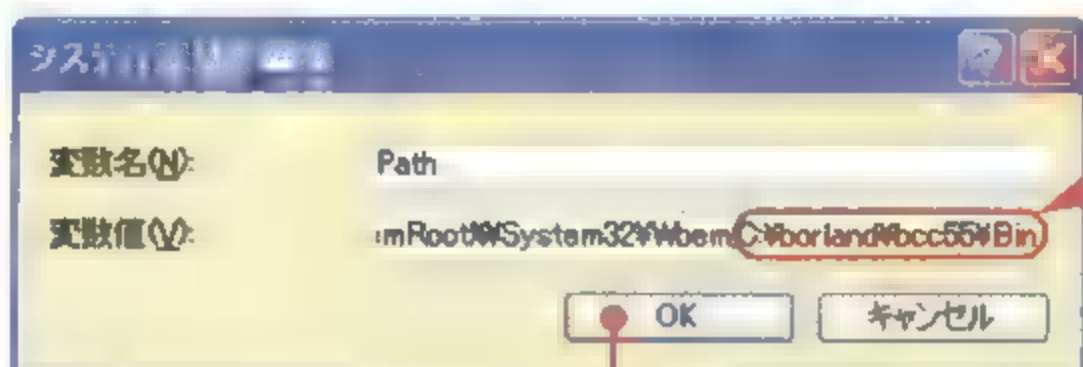
5. 環境変数の設定

インストールが完了したら、次に環境変数（PATH、パス）の設定を行います。環境変数を設定するには、<コントロールパネル>を開いて、以下の手順に従います。

1 <パフォーマンスとメンテナンス>をクリックして、

3 <システムのプロパティ>ダイアログボックスが表示されます。

4 <詳細設定>をクリックし、

5 <環境変数>をクリックします。

6 <環境変数>ダイアログボックスが表示されるので、

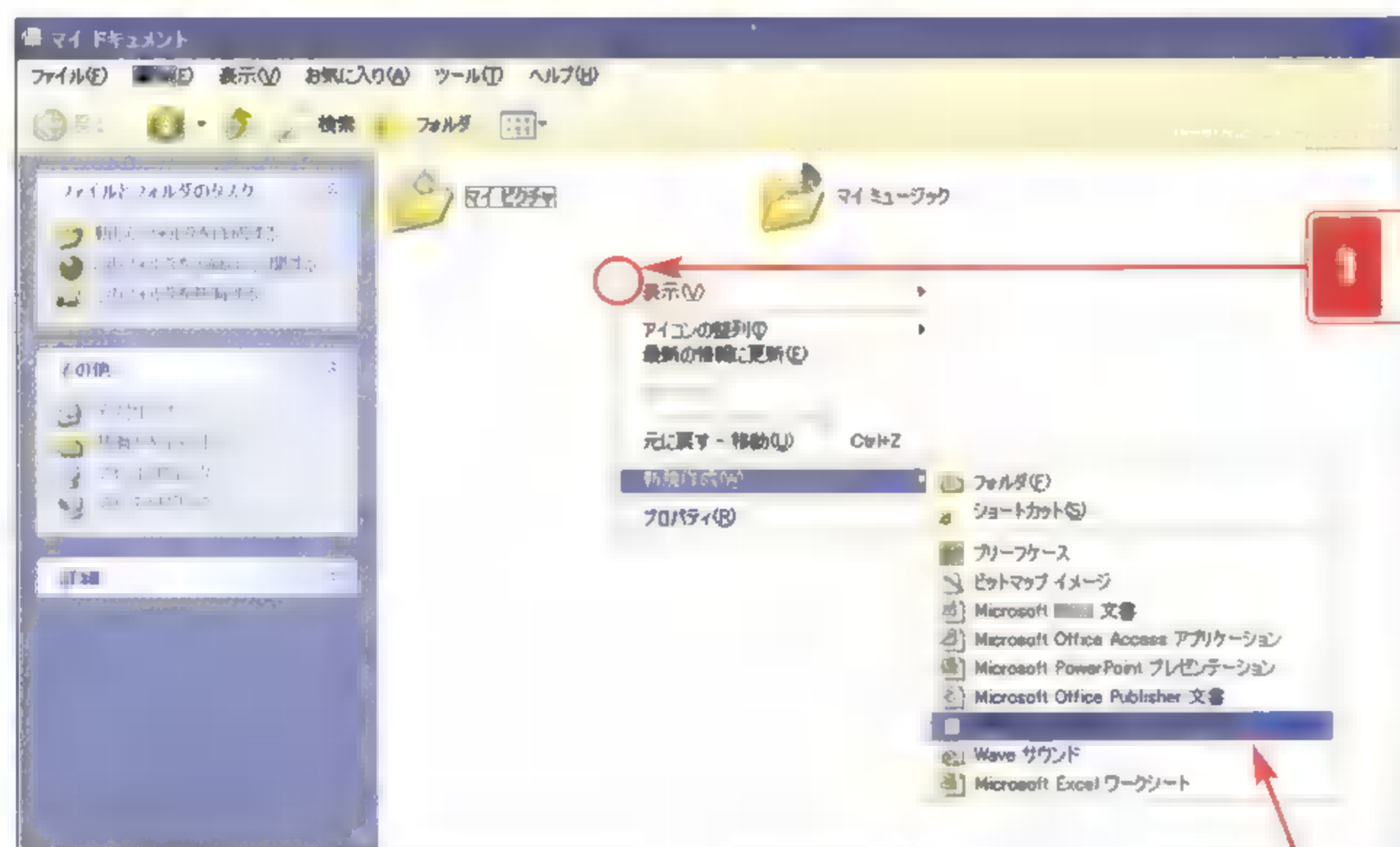
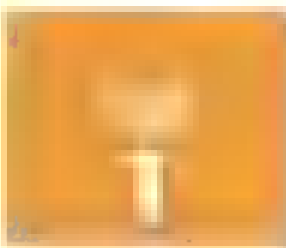
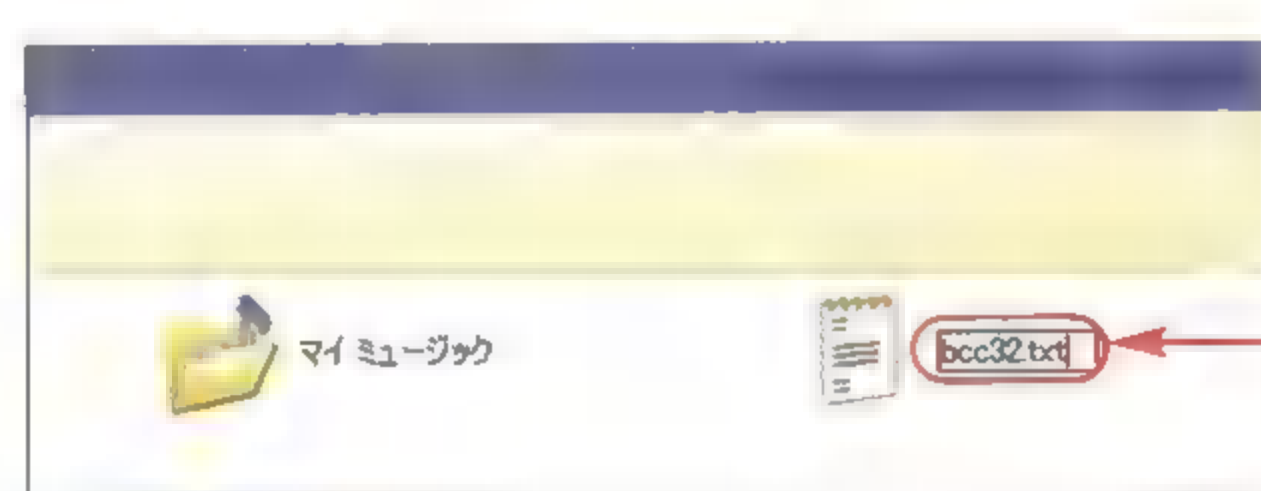
7 <変数>の中から<Path>を選択し、

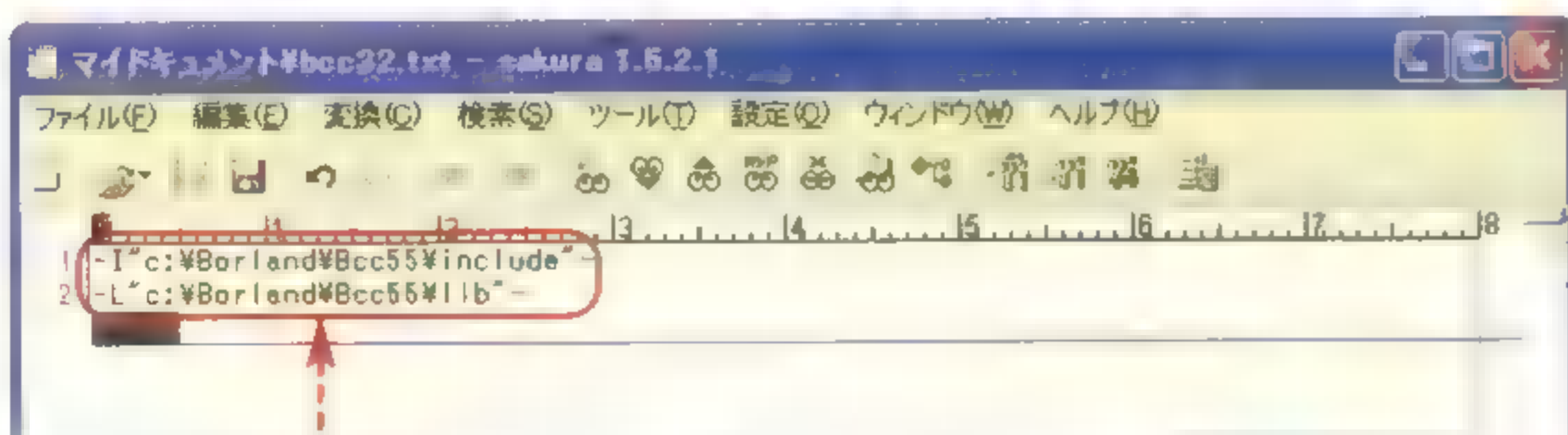
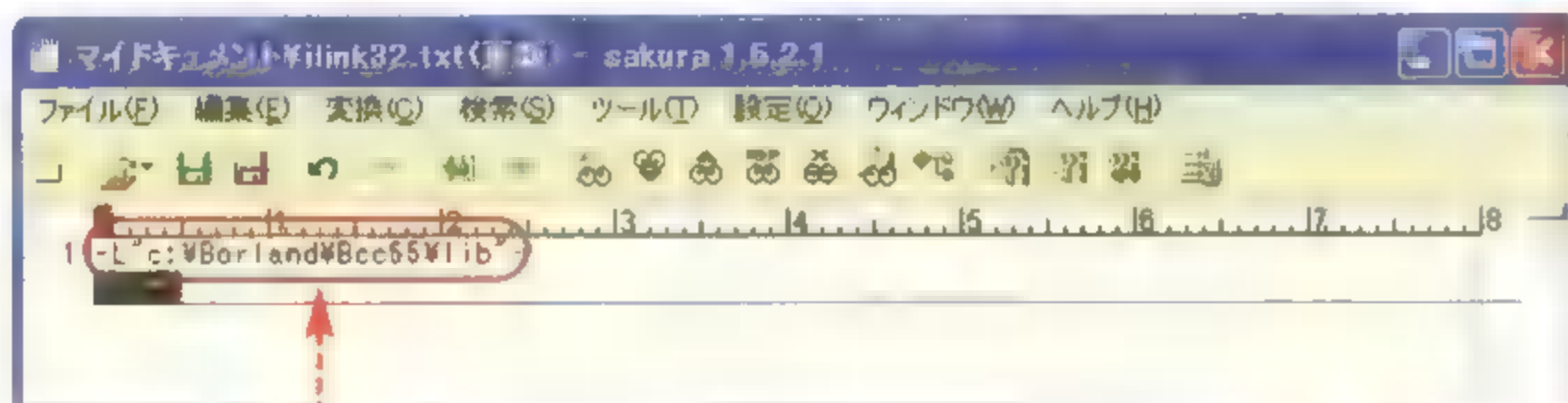
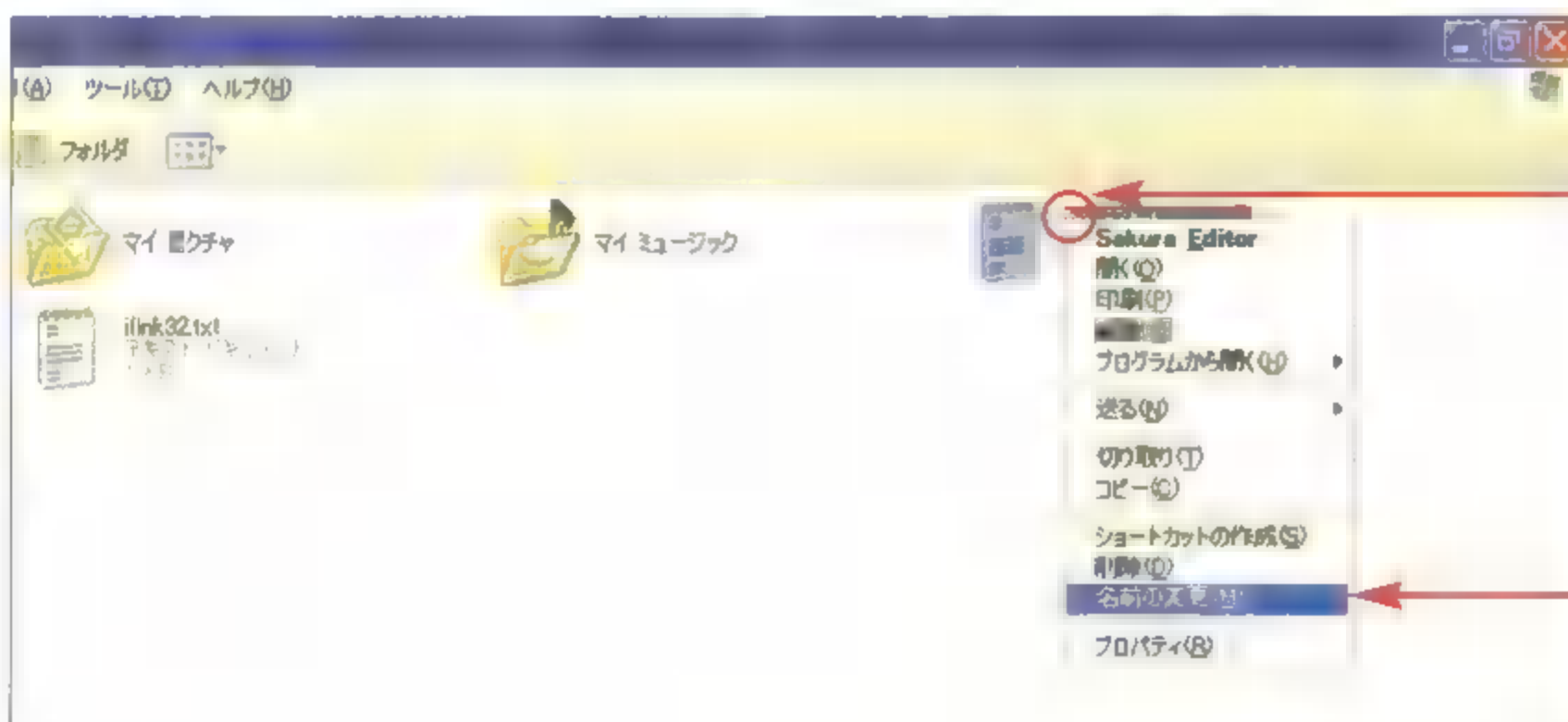
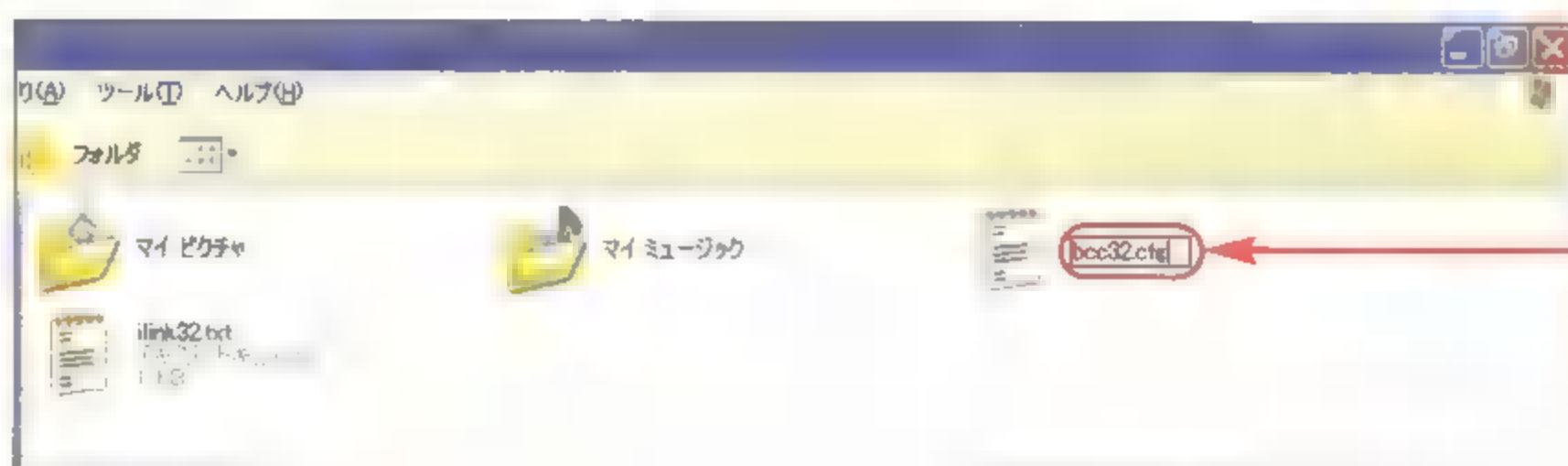
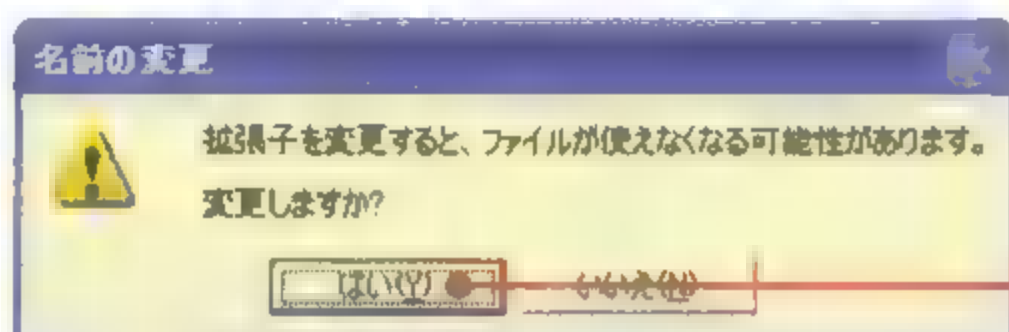
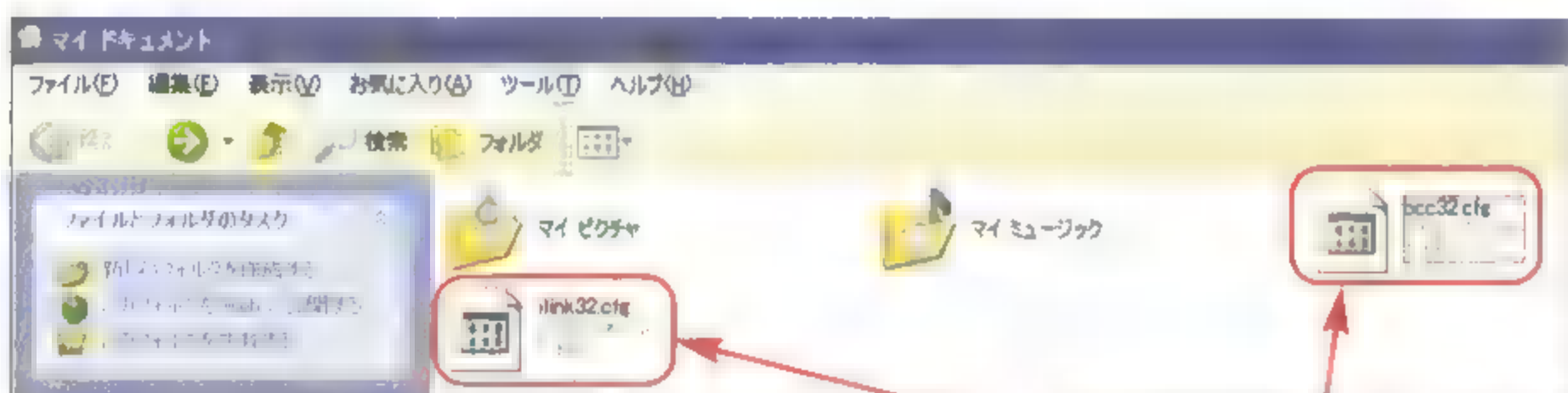
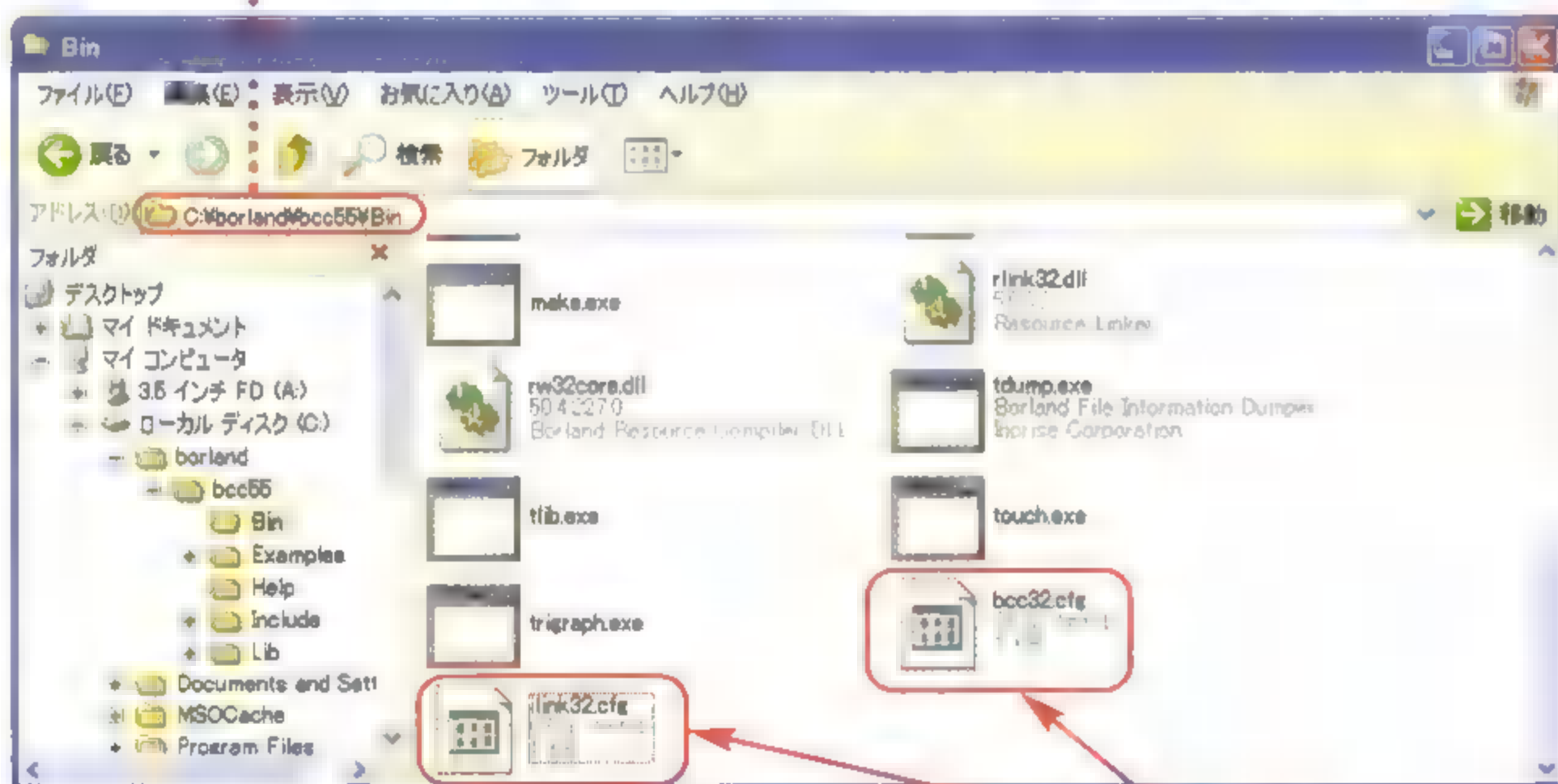
8 <編集>をクリックします。

9 <システム変数の編集>ダイアログボックスが表示されます。

10 <変数値>の最後に、「;C:¥borland¥bcc55¥Bin」を追加し、

追加する文字列の先頭に「;(セミコロン)」を入れるのを忘れないようにしてください。

インストール先を変更した場合は、「インストール先のフォルダ¥bcc55¥Bin」となるように入力してください。

11 <OK>をクリックします。

12 表示されているすべてのダイアログボックスの<OK>をクリックすると、環境変数の変更内容が保存されます。

6. cfgファイルの作成

最後に、コンパイラの設定を記載するcfgファイルを作成します。ここでは、まずテキストファイルでcfgファイルの内容を作成し、後でそのテキストファイルの拡張子をcfgに変更する方法を解説します。これ以外の方法で作成しても問題ありません。

まずテキストファイルを作成するために適当なフォルダを開き、次の手順に従います。

同様の手順で「ilink32.txt」を作成し、
そのファイルを開きます。

14 作成した2つのcfgファイルを、C:¥borland¥bcc55¥Binフォルダに移動します。

これで、コンパイルを行うための準備が完了しました。

付録 2

- ASCII
- 制御コード
- 改行のキーコード

文字コード

コンピュータで文字を表現するためには、文字コードを利用します。1文字を1バイトで表現し、1バイトは256通りの値が表現ができるため、値1つに文字1つを割り当てます。ここでは、どの数値にどの文字が割り当てられているかを一覧にして示します。

1. ASCIIコード

「ASCII」とは、American Standard Code for Information Interchangeの略で、英数字や記号のコンピュータ世界共通の文字コードです。

ASCIIコードでは、文字1つを1バイトで表現します。1バイトは256通りの値が表現でき、アルファベットは26文字ですので、大文字と小文字や記号を合わせても1バイトで十分にすべての文字を表現できます。

また、ASCIIコードでは0x00～0x1fと0x7fに「制御コード」が割り当てられています。制御コードとは、たとえば「改行」のような目に見えない文字などを表します。入門レベルのプログラミングでは、まず必要とはなりません。唯一「改行」の文字コードは利用するかもしれませんので、「**改行の文字コードは0x0a**」と覚えてしまってください。

ASCIIコードで表現できる英数字や記号は、次のとおりです。

下位4ビット＼上位4ビット	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0							制御コード									
1																
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	¥	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Column getch()関数とkbhit()関数

scanf()関数や**gets()**関数では、最後にEnterキーが押されるまで入力処理が開始されません。入力されたキーを即座に取得したい場合は「**getch()**関数」と「**kbhit()**関数」を組み合わせて利用します。ただしこれらの関数はANSI規格ではありませんので、コンパイラによっては用意されていない可能性があります。

kbhit()関数は、「関数が実行されたとき、キーボードが押されている」ことを確認する関数で、押されていれば戻り値として「1」を、押されていなければ「0」を返します。

getch()関数は、「今押されているキーを取得する」関数です。たとえばAのキーが押されている場合には、戻り値として'A'を返します。また、**getch()**関数でキー入力を受け取った場合、ユーザーが入力したキーは画面に表示されません。

これらを利用して入力したキーコードを表示するプログラムは、次のようになります。

getkey.c　押されているキーの取得

```
01 #include <stdio.h>
02 #include <conio.h>
03
04 int main()
05 {
06     printf("何か入力してください。¥n");
07     /* キーボード入力を受け取る */
08     while( 1 ) {
09         if( kbhit()==1 ){
10             int c = getch();
11             printf("入力したキーのキーコードは%xです。¥n", c);
12             break;
13         }
14     }
15     return 0;
16 }
```

- 02行目: getch()関数などを利用するためには、このヘッダーファイルをインクルードします。
- 08行目: 無限ループで、キーが入力されるのを待ちます。
- 09行目: キーが入力されると、処理がif文の中に入ります。
- 10行目: 押されているキーを取得します。
- 12行目: 無限ループを抜けます。

```
C:¥work>getkey
何か入力してください。
入力したキーのキーコードは41です。
C:¥work>
```

付録
3

覚えておきたいキーワード
- 迷路探索
- プログラムの仕様

プログラムの応用例

C言語の基礎的なプログラミングが理解できたら、次のステップは「今までより一歩高度なソースコードを読む」ことを試してみましょう。プログラムがどのように動作するのかをソースコードから読み解くことで、より深い理解を得ることができます。

1. プログラムの仕様

■ サンプルプログラムの概要

今までよりも一段階難しいプログラム例として、ここでは「迷路探索」のプログラムを作成します。今までのプログラムよりかなりコード量が多く、また処理もやや複雑なので理解しにくいかもしれません。重要な部分をあらかじめ説明しますので、あきらめずに根気強くコードを読んでみてください。

■ サンプルプログラムの仕様

このプログラムは再帰関数（Sec.19参照）を利用して、迷路のスタートからゴールまでの経路を探索します。具体的には、ゴールにたどりつくまで次の手順を繰り返して探索を行います。

ここで、迷路は正方形のマス目を並べた形状で、4×4の大きさであるとします。迷路の各マス目からは上下左右の4方向にのみ動けますが、カベがある方向には動けないものとします。

(1) 現在のマス目がゴールかどうかを調べ、ゴールであれば処理を終了する。

(2) 現在のマス目から移動できる方向を探す。

(3) 移動できる方向は、自分が通ってきた方向ではないことを調べる（2つのマス目をいったりきたりしないため）。

(4) 移動できない（カベにぶつかる）場合は、画面に「×」を表示して、経路を1つ戻る。

(5) 移動できる方向へ移動し、どこへ移動したかを表示する。

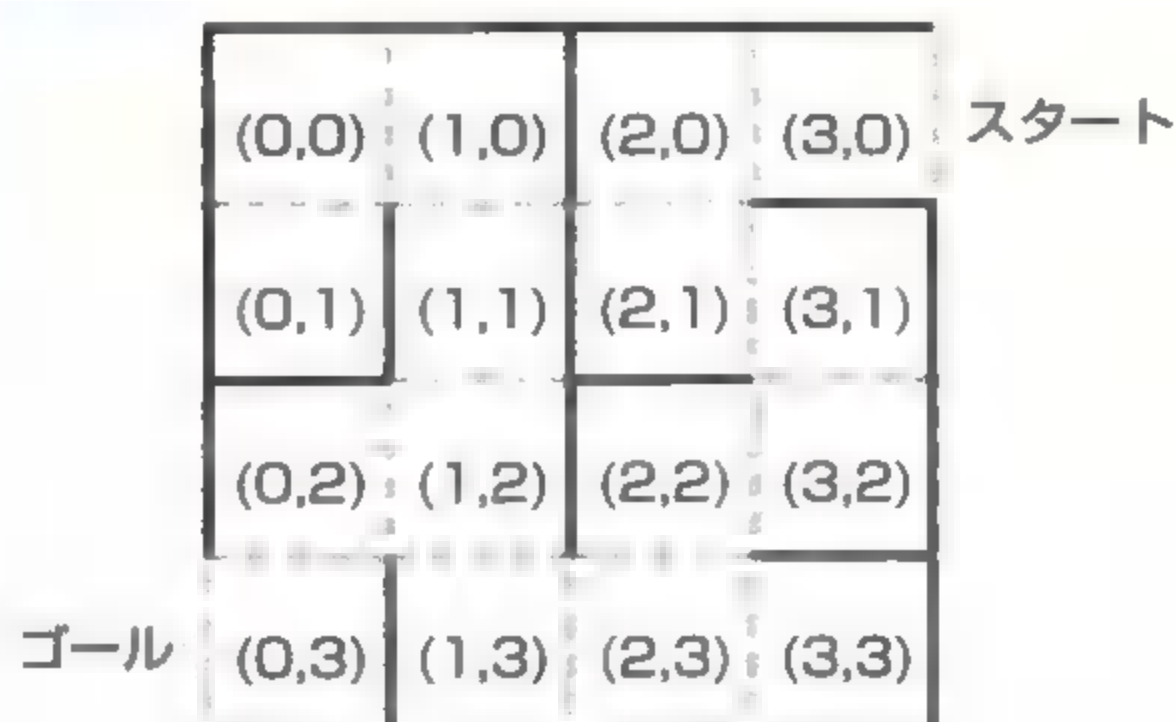
(1)〜(5)の手順を再帰によって繰り返します。ゴールにたどりついたら関数の呼び出しを終了し、再帰関数のすべての呼び出しを抜けます。

2. 迷路データ

■ 迷路の概要

このプログラムでは、次の図のような迷路をデータとして持ちます。

スタート地点は迷路の右上のマス目(3, 0)とし、ゴール地点は左下の(0, 3)からさらに左に1歩動いた(−1, 3)とします。

図1　迷路の全体図

■ 迷路データの保存

迷路のデータは、**unsigned int**型のグローバル変数**maze**に保存します。**maze**は迷路と同様に4×4の2次元配列になっており、各マス目の「どちらの方向にカベがあるのか」という情報を持ちます。たとえば、(0, 0)のマス目のどちらの方向にカベが存在するかを調べるには、**maze[0][0]**を参照するとわかります。

■ 迷路データの仕様

このプログラムでは、迷路データとして、迷路の各マス目の上下左右のどの方向にカベがあるかを持ちます。カベの有無の情報は「ビット」で表します。カベの上下左右にそれぞれビットを割り当て、そのビットが1であればカベがあり、0であればカベがないと判断します。

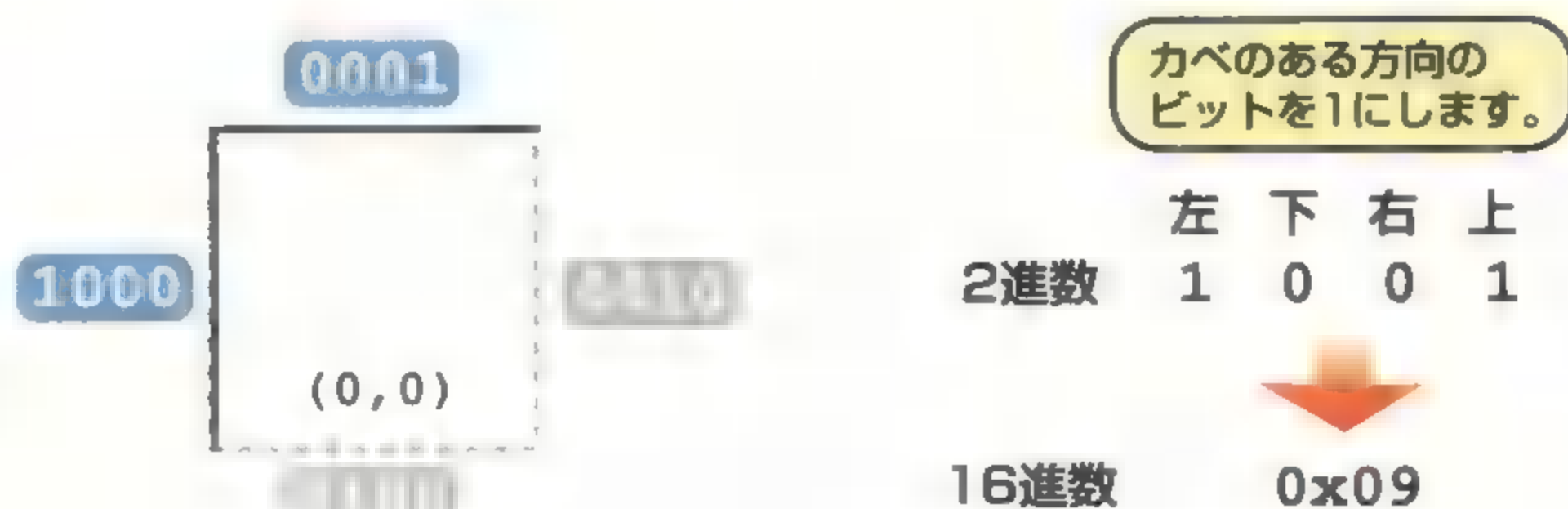
具体的には、次のようになります。

方向ごとに別のビットを割り当てているので、複数の方向にカベがある場合も、カベのある方向のビットを別々に1として表現することができます。

たとえば、迷路の図の(0, 0)のマス目は上と左がカベです。この場合、「0001(0x01)」と「1000(0x08)」のビットをオンにする(1に設定する)ので、「1001(0x09)」となります。

表1　カベの有無情報のビット

方　向	ビット	16進表記	意　味
上	0001	0x01	上方向にカベがあります。
右	0010	0x02	右方向にカベがあります。
下	0100	0x04	下方向にカベがあります。
左	1000	0x08	左方向にカベがあります。

図2 ビットによるカベの表現

3. 関数の仕様

■ 引数や戻り値

このプログラムには、次の2つの関数があります。

(1) next_cell()関数

再帰を利用して、迷路探索を行います。

(2) main()関数

next_cell()関数を呼び出す準備などを行います。

ここでは**main()**関数については説明を割愛して、**next_cell()**関数についての説明を行います。まず、**next_cell()**関数の引数や戻り値は、次のとおりです。

構文 next_cell()関数

```
int next_cell(int x, int y, int dir);
```

(1) 処理概要

現在地を(x, y)として、次に進める方向を探します。進める方向がある場合は、再帰してさらに先に進めるかどうかを探し、いき止まりになったら0を返します。

(2) 引数

引数は次のとおりです。

表2　next_cell()の引数

引数名	意　味
x	進める方向を探す基準のマス目のx座標を指定します。
y	進める方向を探す基準のマス目のy座標を指定します。
dir	どちらの方向から進んできたかを示します。 2つのマス目の間をいったりきたりしないために、 次の値を利用して方向を指定します。 上　0 右　1 下　2 左　3

(3) 戻り値

いき止まりになったら、0を返します。ゴールにたどりついたら、1を返します。

■ 処理の詳細

では、**next_cell()**関数の処理の詳細について説明しましょう。

next_cell()関数は、特定のマス目から進める方向かあるかどうかを探す関数です。マス目は、引数の**x**と**y**を利用して、(**x**, **y**)の形式で指定します。これを「現在地」と呼びます。

関数の処理は、次の順に行われます。

(1) 現在地を表示する。

(2) もし現在地がゴールであれば、戻り値として1を返す。

(3) 上→右→下→左の順番に、繰り返し処理で以下の(4)～(7)を行う。

(4) カベがあるかを調べ、カベがあったら次の方向を調べる。

(5) カベがない場合、引数**dir**を利用して、現在地に移動してきた元の位置(経路上の1歩手前)でないかを調べる。移動してきた元の位置であれば、次の方向を調べる。

(6) 移動できる方向の座標を**next_cell()**関数に与えて呼び出す(再帰)。

(7) 再帰で呼び出した**next_cell()**関数の戻り値が1であれば、ゴールに到達したという意味になるため、(3)の繰り返し処理を抜ける。

(8) 上下左右をすべてチェックした後、**next_cell()**関数からの戻り値が0(もしくは進めるマス目がない)場合、画面に「×」を表示する。

(9) 再帰によって呼び出した**next_cell()**関数の戻り値(一度も呼び出さなかった場合は値0)を、そのまま返す。

この処理手順をまとめて簡単にいうと、**next_cell()**関数とは、

- 進めるマス目には、まず移動してみる。
- そのマス目からさらに移動できる場合、さらに移動する。
- いき止まりになるまで繰り返し、いき止まりになったら0を返す。
- **next_cell()**関数から0が返された場合、その方向はすべていき止まりであるとわかる。この場合には、違う方向を探す。
- 違う方向に進めない場合、そのマス目から移動できるセルはすべていき止まりであるとわかる。この場合には、0を返す。
- 1が返されたら、どこかでゴールにたどりついたということを意味する。この場合、1を返す。

という処理を繰り返している関数です。

4. カベの有無の調べ方

カベの有無を調べるコードを抜き出してみましょう。

```
/* 上→右→下→左の順番にカベがないか調べる */
for(i=0; i<4; i++) {
    /* ビットが1かどうかを調べる */
    if( (maze[y][x] & (1 << i)) != 0 ) {
        ⋮
```

少し複雑ですが、次の2点を押さえればどのような処理を行っているかがわかります。

(1) ビットシフト

```
(1 << i)
```

このコードは、「数値1を左に**i**ビットシフトする」という意味です。**for**文による繰り返し処理であるため、**i**の値は繰り返し処理の回数によって増えていきます。すなわち、上のビットシフト演算の結果は、次のようになります。

表3　ビットシフトと繰り返し処理

処理回数	iの値	演算結果(2進数)	方向
1	0	0001	上
2	1	0010	右
3	2	0100	下
4	3	1000	左

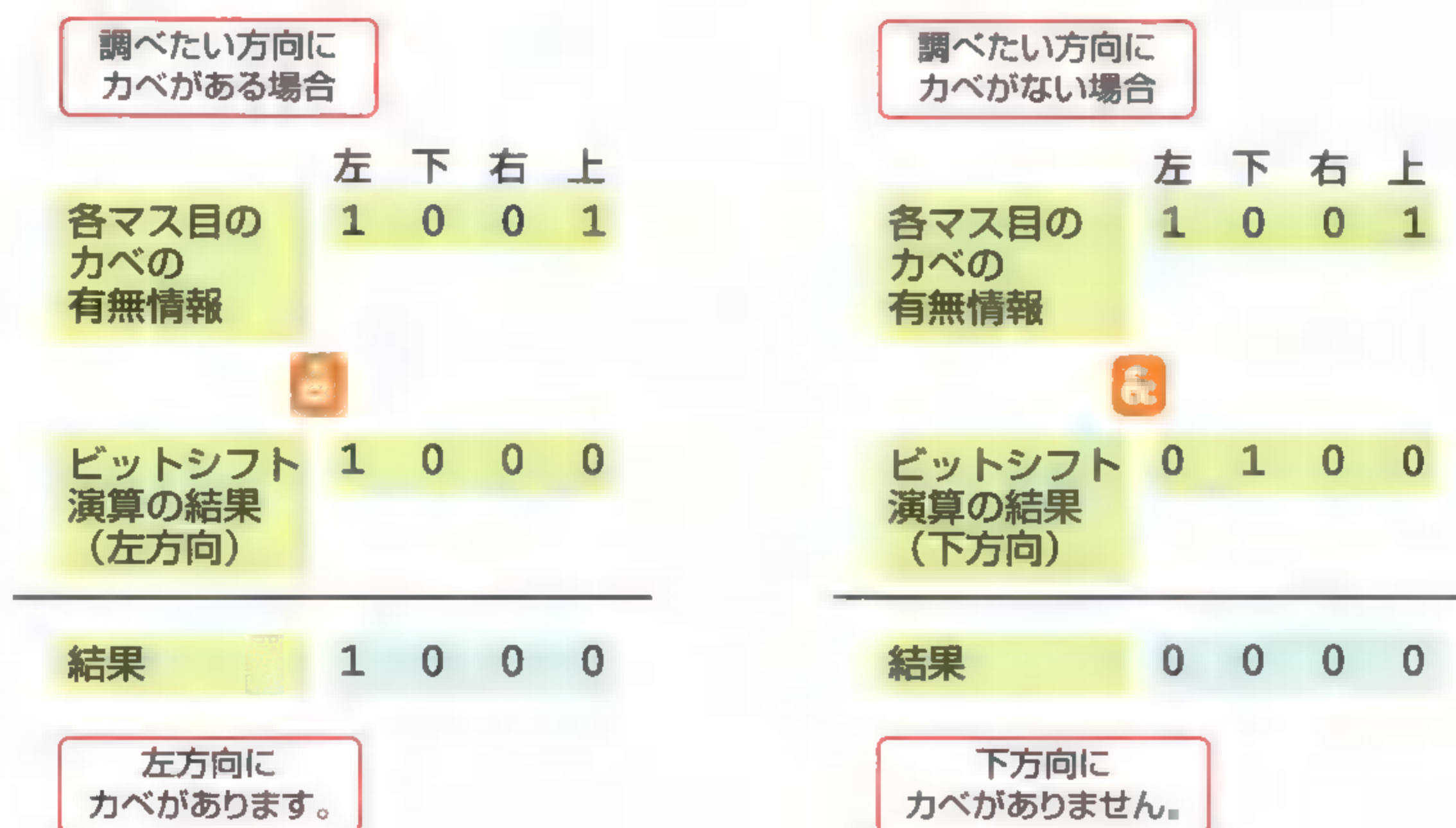
つまり、このビットシフト演算の結果は、そのまま「カベの有無データ」のビットに対応しています。繰り返し処理において**i**を左に1ビットずつシフトしていき、このビットシフト演算の結果とマス目のカベの有無情報を順番にチェックすることにより、上、右、下、左の4方向のカベの有無を判断することができます。

```
(maze[y][x] & (1 << i)) != 0
```

(2) &演算子

続いて、ビットシフト演算の結果と、配列**maze**のデータを&演算子で論理積をとることで、指定したマス目の各方向にカベがあるかどうかを調べます。

ビットシフト演算の結果と配列**maze**の論理積（AND）をとった結果が1になるということは、「今カベがあるか調べたい方向」について「配列**maze**の迷路データの該当ビット」がともに1であるという意味になります。つまり、論理積（AND）演算の結果が0以外になる場合は、今調べようとしている方向について迷路データ上にはカベがあるということです。

図3　カベ情報との論理積演算

5. ソースコード

以上の説明を踏まえた上でソースコードを読んでみましょう。迷路探索を行うプログラムは、次のようになります。

maze.c 迷路探索

```
01  #include <stdio.h>
02
03  /* プロトタイプ宣言 */
04  int next_cell(int x, int y, int dir);
05
06  /* 迷路データ 1がカベ、0は通路 */
07  unsigned int maze[4][4] = {
08      {   0x09,   0x03,   0x09,   0x07 },
09      {   0x0E,   0x0A,   0x0C,   0x03 },
10      {   0x09,   0x02,   0x09,   0x06 },
11      {   0x06,   0x0C,   0x04,   0x07 }
12  };
13
14  int main()
15  {
16      int ret;
17      /* スタート地点(迷路の右上から) */
18      int x = 3;
19      int y = 0;
20
21      printf("迷路探索スタート¥n");
22      ret = next_cell(x,y,1);
23      if( ret == 1 ) {
24          printf("迷路探索を終了します。¥n");
25      }
26
27      return 0;
28  }
29
30  /* 現在地(x, y)から、次に進める方向を探す関数 */
```

次に進めるマスを探します（右には戻らないよう指定しています）。

next_cell()関数を呼び出して1が返った場合、ゴールに到達しているということです。

```
31  int next_cell(int x, int y, int dir)
32  {
33      int i;
34      int ret = 0;
35      printf("(%d,%d)→", x, y);
36      /* もし現在地がゴールであれば、1を返す */
37      if( (x==-1) && (y==3) ){
38          printf("ゴール!¥n");
39          return 1;
40      }
41      /* 上→右→下→左の順番にカベがないか調べる */
42      for(i=0; i<4; i++){
43          /* ビットが1かどうかを調べる */
44          if( (maze[y][x] & (1 << i)) != 0 ) {
45              /* カベがあるので次を探す */
46              continue;
47          }
48          /* カベはないが、通ってきた方向ではないか? */
49          if( i == dir ) {
50              continue;
51          }
52          /* 次のセルを探す */
53          switch(i){
54          case 0:    /* 上に移動できる場合 */
55              ret = next_cell(x,y-1,2);
56              break;
57          case 1:    /* 右に移動できる場合 */
58              ret = next_cell(x+1,y,3);
59              break;
60          case 2:    /* 下に移動できる場合 */
61              ret = next_cell(x,y+1,0);
62              break;
63          case 3:    /* 左に移動できる場合 */
64              ret = next_cell(x-1,y,1);
65              break;
66          }
67          if(ret == 1){
68              break;
```

現在地を表示します。（35行目）

次のマス目として、現在地より「1つ上」、通ってきた方向に「下」を指定します。（55行目）

同様に、現在地より「1つ右」、通ってきた方向に「左」を指定します。（58行目）

同様に、現在地より「1つ下」、通ってきた方向に「上」を指定します。（61行目）

同様に、現在地より「1つ左」、通ってきた方向に「右」を指定します。（64行目）

ゴールにたどりついたら処理を抜けます。（67行目）

```
69              }
70          }
71          if( ret == 0 ){
72              printf("× ");
73          }
74          return ret;
75  }
```

いき止まりであれば、「×」を表示します。

```
C:¥work>maze
迷路探索スタート
(3,0)→(2,0)→(2,1)→(3,1)→(3,2)→(2,2)→(2,3)→(
3,3)→× (1,3)→(1,2)→(1,1)→(1,0)→(0,0)→(0,1)
→× × × × (0,2)→(0,3)→(1,3)→ゴール！
迷路探索を終了します。

C:¥work>
```

実際にこのプログラムをコンパイルして実行してみましょう。迷路の右上からスタートして、ゴールにいきつくまでにたどった経路がすべて表示されています。この結果から、正しく経路を選択できていることがわかります。

さらに、このプログラムの動作がある程度わかったら、5×5の迷路の経路を探索するようにソースコードを修正し、理解をさらに深めてみてください。

Column ビット演算と定数ラベル

プログラミングにおいてビット演算は重要なテクニックですが、見ただけではどのような処理を行っているのかがわかりにくい場合があります。たとえばP.251の例で利用したビット演算は、ちょっと見ただけでは、「カベの有無をチェックしている」という処理をなかなか連想できません。

```
if( (maze[y][x] & (1 << i)) != 0 ) {
```

カベの有無をチェックしています（maze.cの44行目参照）。

ここではこのようなビット演算に定数ラベルを利用し、わかりやすいソースコードを作成するテクニックを紹介します。

付録

(1) ビット情報に定数ラベルを利用する

カベの有無をチェックしているコードの中で、もっともわかりにくいのは「1 << i」のビットシフトの部分です。この部分をわかりやすくするために、ソースの先頭に次のような定数ラベルを宣言します。この定数ラベルは、カベの方向に応じて1を何ビット左にシフトするかを表します。

```
#define BIT_UP    0
#define BIT_RIGHT 1
#define BIT_DOWN  2
#define BIT_LEFT  3
```

上の場合は0、右の場合は1というようにカベの方向ごとにシフトするビット数を定義します。

(2) カベの有無チェック用の配列を用意する

next_cell()関数に次のような配列を宣言します。

```
unsigned int wall[] = {  (1 << BIT_UP),      (1 << BIT_RIGHT),
                         (1 << BIT_DOWN),    (1 << BIT_LEFT)
};
```

配列wallの各要素は上下左右にカベがある場合のビット情報を表します。つまり、これまでは繰り返し処理で「1 << i」により各方向にカベがある場合の比較値を計算していましたが、あらかじめ配列の初期値で計算しておくように修正します。この配列を利用して、カベの有無チェックを次のように修正します。

```
if( (maze[y][x] & wall[i]) != 0 ) {
```

このようにすることで、配列wallがカベのある場合に1が設定されるビット情報を表していることや、if文でカベの有無チェックを行っていることがわかりやすくなります。

(3) case文の値を定数ラベルに置き換える

最後に、case文に記述する値を定数ラベルに置き換えます。具体的には、maze.cの54行目から始まるcase文を次のように変更します。

```
switch(i){
case BIT_UP: /* 上に移動できる場合 */
  ⋮
```

値「0」を、「BIT_UP」に置き換えます。

このように定数ラベルを利用することで、仮にコメントがなくても「上方向への移動だな」というように、コードの意味がわかるようになります。
ビット演算を行う場合、定数ラベルを利用して「ただのビット情報」に名前を付けることで、読んでいて意味のわかりやすいソースコードを作成することができます。

Index

記号

数字

G

H・I

J・K・L

M・N

O・P

R・S

T・U

V・W・X

Index

Index

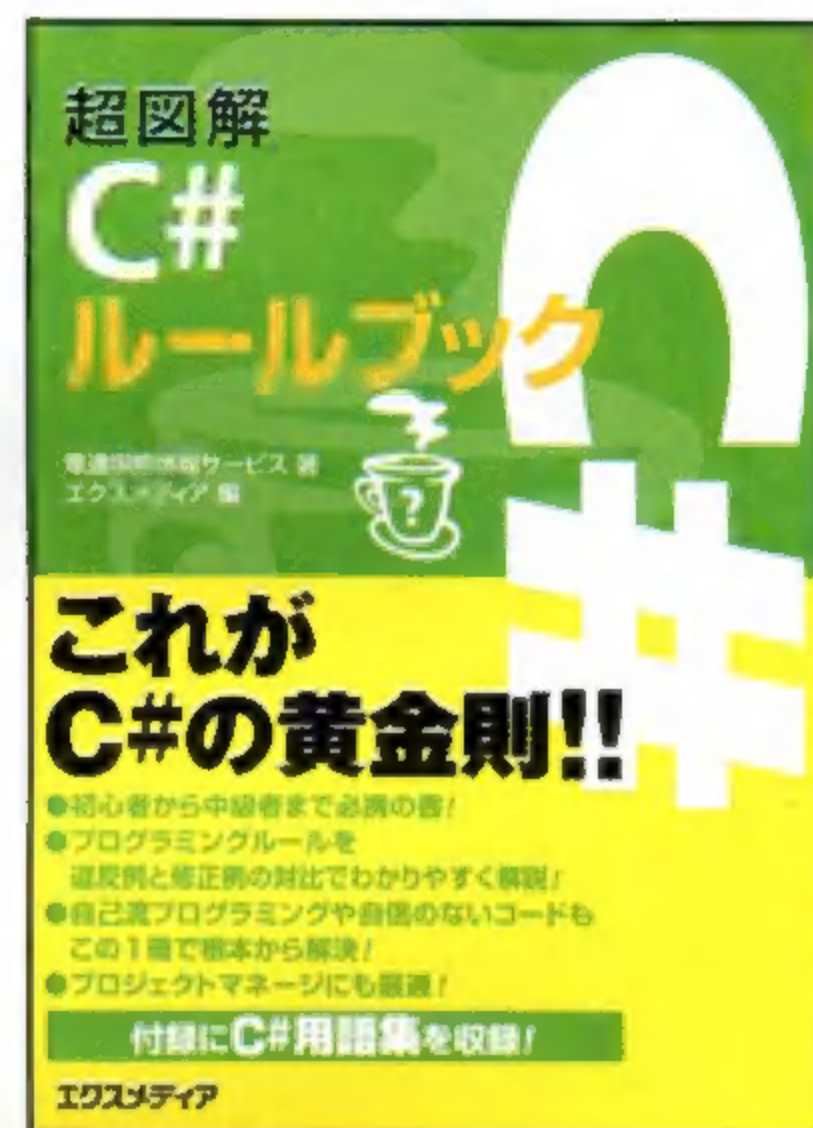
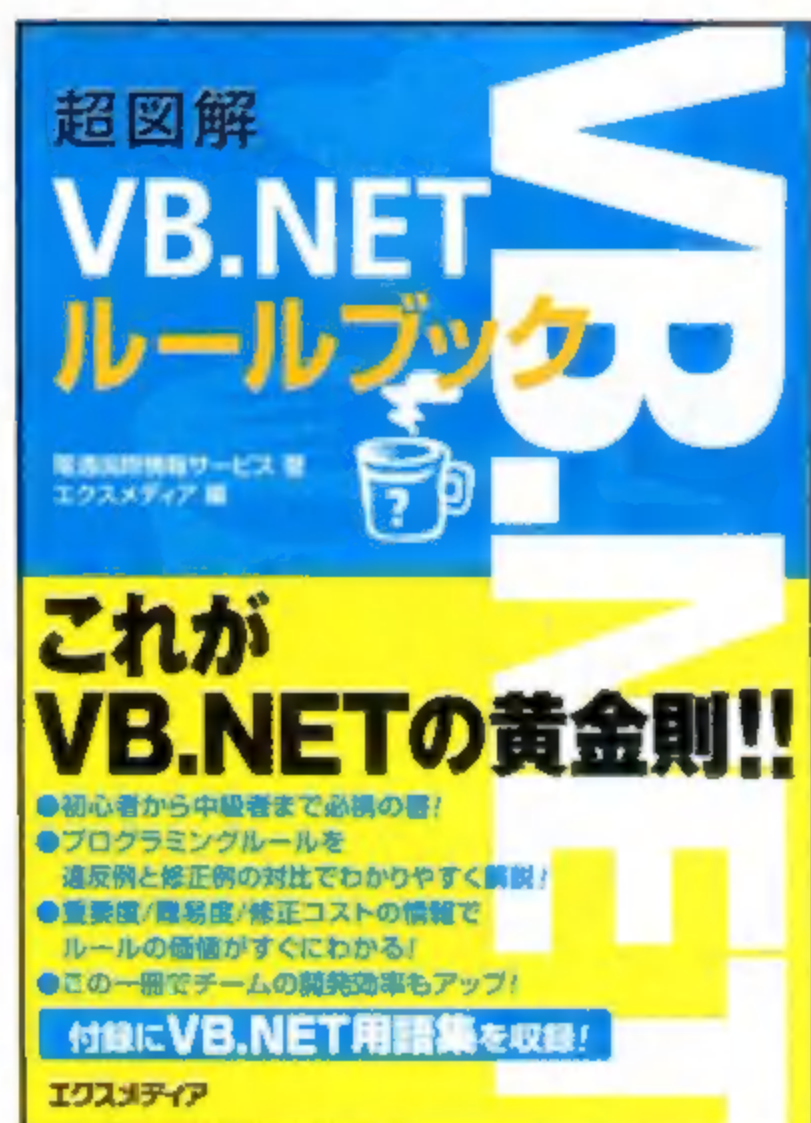
■DTP 近藤しのぶ

ビジュアルラーニング C言語入門

2005年 7月 26日 初版発行

著者 さかお まい

発行者 工藤 俊
発行所 株式会社エクスメディア
〒102-0075
東京都千代田区三番町6-2 三番町弥生館
販売 TEL 03(5215)9034
FAX 03(5215)7751
印刷 三松堂印刷株式会社

ISBN4-87283-441-0

●個人情報の利用について

読者サポートセンター、エクスメディアクラブなどでいただきました個人情報は、原則として下記以外の目的で利用いたしません。下記以外の目的で利用する際は、別途に当社のサービス上でお知らせいたします。

・サービス向上のための参考資料　・賞品を送るための連絡先　・ご質問内容の回答手段

エクスメディアクラブ
X-MEDIA CLUB

このクラブはエクスメディアの書籍愛読者以外にも、電子メールアドレスを持っていて、同クラブの規約にご賛同いただける方に、当社独自のサービスをご提供いたします。「エクスメディアクラブ」の会員として登録すると、会員だけのサービスとして次のような特典が受けられます！

ポイントシステムによる希望書籍のプレゼント

ご購入いただいた当社書籍に付属している「愛読者アンケートはがき」をご返送ください。購入した書籍の金額に応じてポイントをためることができます（「愛読者アンケートはがき」には、必ず会員番号をご記入ください。会員番号の記入がない場合にはポイントが加算されません）。
加算されるポイントは、書籍本体価格の10％（100円未満は切り上げ）。ためたポイントは"1ポイント＝1円"として、当社出版の書籍と交換できます！

◆ポイントをためる
例）『超図解 Excel 2000 for Windows 基礎編』
1,380円購入の場合→140ポイント獲得（下1桁切り上げ）
◆希望書籍と交換
例）ポイントが1,500ポイントたまったところで、本体価格1,359円の書籍と交換した場合
1,500−1,359＝141ポイント 残ります。

GET POINT AND EXCHANGE FOR BOOKS

2 新刊案内などの電子メール配信サービス

新刊案内や会員の方だけのお得な情報など、つねに新しい情報を電子メールで配信いたします。

ご入会金・年会費など一切ございません。
クラブへの入会方法や保有ポイント数の確認、書籍引き換えの申し込みなどの詳しいお問い合わせは、下記URLのホームページをご参照ください。
http://www.x-media.co.jp/CLUB/

書籍通信販売サービス
BOOKS DELIVERY SERVICE

弊社書籍が書店店頭で見つからない場合や、サンプルファイルを収録したCD-ROMやフロッピーディスクを購入ご希望の方は、弊社の書籍通信販売サービスをご利用ください。

※サンプルファイルの価格は、メディアや書籍により異なるため、あらかじめお問い合わせください。なお、PhotoshopおよびIllustrator関連書籍のサンプルファイルは、弊社ホームページからのダウンロードサービスのみになります。

お問い合わせ先

通販直通　TEL 03-5215-7715

ホームページ
HOMEPAGE

http://www.x-media.co.jp/

弊社のホームページでは、これまでに制作したすべての書籍を紹介しています。書店に書籍がない場合などに、内容確認にご利用になられると便利です。また、一部例外を除き書籍内で使用したサンプルデータをアップロードしていますので、書籍と同様の操作を試したいときなどにご利用ください。

新刊書籍案内
弊社書籍の最新刊の内容について、目次や索引、ページのサンプルなどの詳しい情報を掲載しています。最新刊はどんな内容かを知る上で、優れたガイド役になってくれることでしょう。

サンプルデータ
ここでは書籍内で使用したサンプルデータをダウンロードできます（一部例外を除く）。書籍に掲載されているものとまったく同じデータを使い、ご自分のパソコンで実際に同じ手順で操作すれば、機能を理解するスピードも全然違います。

書籍案内
エクスメディアが制作したすべての書籍の内容を参照することができます。「シリーズ別リスト」、「内容別リスト」、「出版社別リスト」の3つの分類項目から、目的の書籍を確実に探し出すことができます。

書籍 FAQ
「愛読者アンケートはがき」や「ご意見箱」に寄せられた数多くのご質問の中から、頻繁に尋ねられるご質問に関する回答を掲載しています。

内容訂正
エクスメディアでは、細心の注意を払って原稿の校正と修正を重ね、よりクオリティの高い解説書を制作することに日夜努力をしていますが、万が一記述に誤りが発見された場合は、このコーナーに掲載しています。現在、「版ごと」に訂正内容を掲載しています。

ご意見・お問い合せ
各コーナーに関するご意見やご感想、ご質問などを受け付けています。また、ホームページだけでなく、弊社書籍に関する声もお寄せください。

エクスメディアのシンボルマークは、人にやさしく情報を伝える「情報デザイン企業」という当社の企業理念をビジュアライズしたもので、社名のイニシャルである、未知数「X」をモチーフにデザインしたものです。
正方形の対角線上に、シャープなラインをクロスさせることで「X」を表現しています。このシャープなX-ラインは、情報そのものを意味したもので、複数のラインがクロスするところは、エクスメディアとユーザーの出会いのときめきと、信頼のコミュニケーションを表わしています。
また、X-ラインは、企業姿勢として、常に新しいメディアを切り拓いて行く先進性と創造力、そして知的でダイナミックな行動力を表わしています。
背景にある手描きの躍動感あるレインボーカラーは、エクスメディアの事業イメージの広がりと、豊かな夢とロマンを表現したものであり、ユーザーに対しては発信する情報のコンテンツそのものに活力があり、楽しさと親しみやすさ、わかりやすさを持ったヒューマンなものでありたいと、願う気持ちを表わしたものです。

カバーデザイン：近藤　しのぶ

ISBN4-87283-441-0
C3055 ¥2000E

9784872834413

定価：本体2,000円＋税

1923055020009

ビジュアルラーニング
C言語入門

さかおまい　著

エクスメディア